版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择脱裤滚蛋裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不能秒看你得到了知识，所以请不要得到书籍后就觉得法百自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择脱裤滚蛋裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不能秒看你得到了知识，所以请不要得到书籍后就觉得法百自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

版权所有侵犯必究 SHEN YE SHU HUI

O'REILLY

Spark
全栈数据分析

Apache Data Science 2.0: Building Full-Stack Data Analytics Applications with Spark

[美] Russell Jurney 著
王道廷 译

版权所有侵犯必究 SHEN YE SHU HUI

电子工业出版社
Publisher of Electronic Industry
北京·BEIJING

版权所有侵犯必究 SHEN YE SHU HUI

内 容 简 介

本书介绍了开源超级计算工具Spark的敏捷数据科学方法论。综合作者在行业中多年的实际工作经验，为数据科学团队提供了一套以实用编程为基础的、以数据为中心的敏捷科学研究方法论。中小型公司之间一些常见工具的使用，包括从前端分析应用程序的Spark和Spark SQL，到把不规整的数据科学原型迭代地转化为生产级别可维护的、可扩展的、分析驱动的应用程序的过程。

本书适合软件工程师、数据科学家、工程师、分析师和数据工作中的专用物流。

©2017 by Data Syndrome LLC
Simplified Chinese Edition, jointly published by of O'Reilly Media,Inc and Publishing House of Electronics Industry,2018 Authorized translation of the English edition,2017 O'Reilly Media,Inc.,the owner of exclusive rights to publish and sell the work.
All rights reserved including the right of reproduction in whole or in part in any form.

本书简体中文版由电子工业出版社与O'Reilly Media,Inc.授予电子工业出版社。未经许可，不得以任何方式复制或抄袭本书的任何部分，专有出版权受法律保护。

版权贸易合同登记号：图字：01-2017-5590

图书在编编目（CIP）数据

Spark 全栈数据分析 / [美] 罗素·尤尼（Russell Jurney）著，王道廷译.
—北京：电子工业出版社，2018.11
书名原文：Apache Data Science 2.0: Building Full-Stack Data Analytics Applications with Spark
ISBN 978-7-121-35346-2

1. ①Spark ② ①罗… ②王…	3. ①数据处理软件 Ⅳ. ①TP274
2. ①大数据 ②数据挖掘	

发现编目：CIP 数据核字（2018）第 227593 号	
责任编辑：刘恰恰	
责任编辑：牛 勇	出版发行：电子工业出版社
排版设计：李 辰	印刷：三河市双峰印刷装订厂有限公司
出版：电子工业出版社	装订：三河市双峰印刷装订厂有限公司
出版发行：电子工业出版社	开本：787×980 1/16 印张：23.5 字数：413 千字
版次：2018 年 11 月第 1 版	
印次：2018 年 11 月第 1 次印刷	
定价：89.00 元	

凡所有权内容之外出版社所有侵权权益，请向相关书店退换，若有异物，请与本社发行部联系，联系方式请电话：（010）88254888，88258888。
质量投诉请发邮件至 zlts@phei.com.cn，盗版投诉请发邮件至 dbqq@phei.com.cn。
本书影响联系方式：010-51260888-819 faqi@phei.com.cn。

版权所有侵犯必究 SHEN YE SHU HUI

O'Reilly Media, Inc.介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O'Reilly一直聚焦着这个社区的发展需求和趋势走向，通路资本打造开放资本市场，而我们的目标是要的技术趋势——通过面对未来而“预测的行动”来阐述自己对新的技术、商业法则、开放技术以及平台搭建的看法。O'Reilly的选择是通了开放源代码、互联网、新兴趋势及技术。

O'Reilly 为软件开发人员带来革新性的“金融牛”，聚焦第一个技术明动（GNN），我们帮了那帮高端技术的人，他们是最早采用开源代码和标准的人，我们帮了Make杂志，从而成为DIY革命的主要先锋；在第一届所形成通过多种形式吸引信息与人的技术，O'Reilly 的会议与峰会集聚了业界最隆重和高端已编程的业领袖，高同越我们的会聚，产业的革命性思想。作为技术人员的信息出流通道，O'Reilly 观点还通过电子定制知识传播普及新的思想，作为技术人员的信息过程提供了一个高效率表述的媒介。每一项O'Reilly 的产品都反映了自己不可或缺的理念——信息是最宝贵的财产之一。

业界评论

“O'Reilly Radar 博客有什够味。”
——Wired

“O'Reilly 充满一点有《本事至少和此想同行》你可能会成为了解官方文化的桥梁。”
——Business 2.0

“O'Reilly Conference 是我看见够思想碰撞的地方对者是。”
——CRN

“一本O'Reilly的书就代表一个好主题，更重要的，它是令行的人。”
——Irish Times

“Tim是个很会做事的有人，他不光成就了自己的，也成了别人的事业。正是”同时他把产业知识传播到Yogi Berra的观点来了，“如果你去路上遇到岔路，走小路（思维）”神秘法则。Tim 知手每一点概念来了小路，而且更无论概念一经传播的路，你要记概念不断。”
——Linux Journal

版权所有侵犯必究 SHEN YE SHU HUI

译者序

这几年，大数据、人工智能都是计算机界的年会重地门的话题。数据科学也越来越受到各公司的关注。我编辑朋的很多不同行业的公司都对大数据与人工智能的应用解新被此，在朋友的引荐下我认识了这位资深中的翻译。可以证明数据分析技术有真正地与它们的商业问题及它们的数据的。我有很多从可以用学习从数据中处理数据。特别的特点是无法取时间为真正的价值，学习数据的要是比较新的技术，大多数公司这些需要和参数。数多公司要的数据了之已的公司不复杂的，最的一些数据科学的平台是有不同。相同是如何以开发工程和数据科学家对相对的工作。把这个数分析工作的一个关键点是流动价值。

本书的重点要数据工作的流程，且通过这本书的工作流程的通道了对不同角色的理解。我的本身也的实际数据分析应用不发行版，在他的帮助之前中，他贡献对这很多公司的是以这些数据分析的工作流程的多种的，这也是我最喜欢的工作，它了到每本种公司进行大数据方面的数据分析。本书是作者多年的经验的总结与升华，涵盖了从团队建立、工作流程到工具选择、经典分析与执行的方法论。本书以一个完整的项目为例，展望全书，展示了敏捷数据科学的整个流程，这也是我最喜欢的部分。这具体数据的发展，比在数据的建设数据变化的基础上的神奇，让步骤的数据科学的步骤，更好地提高开发速度。

Spark 是当前大数据领域最为主流的项目，有着超级 Hadoop/Mapreduce 的性能，可以是大数据领域的事实标准。虽然今天的时间，Spark 所面对的是不同的，Spark 是一个在内存 Hadoop 的生产力大数据处理工具，是一个不但 Spark 相关是编程用户不二选择，不过，Spark 虽然真正是用与高性能的特性，但不代表 Spark 的性能不会出现在任何今天的问题，事实上，随着业务增长，Spark 的性能和效率会成为影响业务的不能接近扩展瓶颈的最大障碍。

这个发展迅速的领域，本书的编写，毫无疑问，应中的作者于 Spark 的知识（以及 Spark）可能已经不有了很多变化，本书的内容也的公司不会大数据的数据分析，还包括的编辑，的的，管理等内容，由于技术不断，难免有些滞后。希望读者能不断的，有编辑之处，不断地与我探讨，我的邮箱是 zwj@jsxsw.com。

王道廷
2018 年夏

版权所有侵犯必究 SHEN YE SHU HUI

版权所有侵犯必究 SHEN YE SHU HUI

仅供非商业用途或交流学习使用

目录

前言 xiv

第Ⅰ部分　准备工作

第1章　理论 3
导论 3
定义 5
方法学 5
敏捷数据科学宣言 6
瀑布模型的问题 10
研究与应用开发 11
敏捷软件开发的问题 14
最终质量：偿还技术债 14
瀑布模型的拉力 15
数据科学过程 16
设置预期 17
数据科学团队的角色 18
认清机遇与挑战 19
适应变化 21
过程中的注意事项 23
代码审核与结对编程 25

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

敏捷开发的环境：提高生产效率 25
用大幅打印实现想法 27
第2章　敏捷工具 29
可伸缩性＝易用性 30
敏捷数据科学之数据处理 30
搭建本地环境 32
配置要求 33
配置Vagrant 33
下载数据 33
搭建EC2环境 34
下载数据 38
下载并运行代码 38
下载代码 38
运行代码 38
Jupyter笔记本 39
工具集概览 39
敏捷开发工具栈的要求 39
Python 3 39
使用JSON行和Parquet序列化事件 42
收集数据 45
使用Spark进行数据处理 45
使用MongoDB发布数据 48
使用Elasticsearch搜索数据 50
使用Apache Kafka分发流数据 54
使用PySpark Streaming处理流数据 57
使用scikit-learn与Spark MLlib进行机器学习 58
使用 Apache Airflow（孵化项目）进行调度 59
反思我们的工作流程 70
轻量级网络应用 70
展示数据 73

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

本章小结 75

第3章　数据 77

飞行航班数据 77

航班准点情况数据 78

OpenFlights数据库 79

天气数据 80

敏捷数据科学中的数据处理 81

结构化数据vs.半结构化数据 81

SQL vs. NoSQL 82

SQL 83

NoSQL与数据流编程 83

Spark: SQL + NoSQL 84

NoSQL中的表结构 84

数据序列化 85

动态结构表的特征提取与呈现 85

本章小结 86

第Ⅱ部分　攀登金字塔

第4章　记录收集与展示 89

整体使用 90

航班数据收集与序列化 91

航班记录处理与发布 94

把航班记录发布到MongoDB 95

在浏览器中展示航班记录 96

使用Flask和pymongo提供航班信息 97

使用Jinja2渲染HTML5页面 98

敏捷开发检查站 102

列出航班记录 103

使用MongoDB列出航班记录 103

数据分页 106

仅供非商业用途或交流学习使用

搜索航班数据 112
创建索引 112
发布航班数据到Elasticsearch 113
通过网页搜索航班数据 114
本章小结 117
第5章　使用图表进行数据可视化 119
图表质量：迭代至关重要 120
用发布/装饰模型伸缩数据库 120
一阶形式 121
二阶形式 122
三阶形式 123
选择一种形式 123
探究时令性 124
查询并展示航班总数 124
提取“金属”（飞机（实体）） 132
提取机尾编号 132
评估飞机记录 139
数据完善 140
网页表单逆向工程 140
收集机尾编号 142
自动化表单提交 143
从HTML中提取数据 144
评价完善后的数据 147
本章小结 148
第6章　通过报表探索数据 149
提取航空公司为实体 150
使用PySpark把航空公司定义为飞机的分组 150
在MongoDB中查询航空公司数据 151
在Flask中构建航空公司页面 151
添加回到航空公司页面的链接 152

仅供非商业用途或交流学习使用

创建一个包括所有航空公司的主页 153
整理半结构化数据的本体关系 154
改进航空公司页面 155
给航空公司代码加上名称 156
整合维基百科内容 158
把扩充过的航空公司表发布到MongoDB 159
在网页上扩充航空公司信息 160
调查飞机(实体) 162
SQL嵌套查询vs.数据流编程 164
不使用嵌套查询的数据流编程 164
Spark SQL中的子查询 165
创建飞机主页 166
在飞机页面上添加搜索 167
创建飞机制造商的条形图 172
对飞机制造商条形图进行迭代 174
实体解析:新一轮图表迭代 177
本章小结 183
第7章 进行预测 185
预测的作用 186
预测什么 186
预测分析导论 187
进行预测 187
探索航班延误 189
使用PySpark提取特征 193
使用scikit-learn构建回归模型 198
读取数据 198
数据采样 199
向量化处理结果 200
准备训练数据 201
向量化处理特征 201
稀疏矩阵与稠密矩阵 203

准备实验 204
训练模型 204
测试模型 205
小结 207
使用Spark MLlib构建分类器 208
使用专用结构加载训练数据 208
处理空值 210
用Route(路线)替代FlightNum(航班号) 210
对连续变量分桶以用于分类 211
使用pyspark.ml.feature向量化处理特征 219
用Spark ML做分类 221
本章小结 223
第8章 部署预测系统 225
把scikit-learn应用部署为网络服务 225
scikit-learn模型的保存与读取 226
提供预测模型的准备工作 227
为航班延误回归分析创建API 228
测试API 232
在产品中使用API 232
使用Airflow部署批处理模式Spark ML应用 234
在生产环境中收集训练数据 235
Spark ML模型的训练、存储与加载 237
在MongoDB中创建预测请求 239
从MongoDB中获取预测请求 245
使用Spark ML以批处理模式进行预测 248
用MongoDB保存预测结果 252
在网络应用中展示批处理预测结果 253
用Apache Airflow(孵化项目)自动化工作流 256
小结 264
用Spark Streaming部署流式计算模式Spark ML应用 264
在生产环境中收集训练数据 265

Spark ML模型的训练、存储、读取 265
发送预测请求到Kafka 266
用Spark Streaming进行预测 277
测试整个系统 283
本章小结 285

第9章　改进预测结果 287

解决预测的问题 287
什么时候需要改进预测 288
改进预测表现 288
黏附试验法：找出黏性好的 288
为试验建立严格的指标 289
把当日时间作为特征 298
纳入飞机数据 302
提取飞机特征 302
在分类器模型中纳入飞机特征 305
纳入飞行时间 310
本章小结 313

附录A　安装手册 315

安装Hadoop 315
安装Spark 316
安装MongoDB 317
安装MongoDB的Java驱动 317
安装mongo-hadoop 318
编译mongo-hadoop 318
安装pymongo_spark 318
安装 Elasticsearch 318
安装Elasticsearch的Hadoop支持库 319
配置我们的Spark环境 320
安装 Kafka 320
安装scikit-learn 320
安装Zeppelin 321

前言

写作本书第 1 版的那段日子里，我刚好因为一次车祸而残疾，每天忍受疼痛折磨，双手也有些不听使唤。当时，一个叫作“职业浏览器”的项目的失败经历正困扰着我，为了从阴影中走出来，我用 iPad 在床上和沙发上写完了本书，尽管那时我的手都没办法切菜了。我在那个项目发布前几周受了伤，还想着坚持把项目做上线，日夜奋战，非常痛苦。在做项目的过程中，我们犯了许多低级错误，让我一直垂头丧气。最终产品糟透了。项目失败的挫折感不时让我难受，而我背部的慢性疼痛更是很少放过我。我的心脏也出了一些问题，心率下降了三分之一，记忆力也出现了衰退。我仿佛进入了一个幽暗的空间，难以找到出路。我要恢复起来，与失败抗争。说来有些奇怪，为了让自己恢复，我写了第 1 版书。我要把我能给团队同事的指导写下来，确保下一个项目成功。我想让自己摆脱这段经历。更重要的是，我想通过帮助别人，让我的人生重新获得意义，不让自己被残疾击垮。这样一件为大众服务以确保其他人不会重复我的错误的好事，我认为是值得去做的。那个失败项目暴露出了一个比我自身的处境更严重的问题，那就是大多数研究都停留在纸面上，从未让能够获益的人实际使用到。这本书就是一剂良方，是应用性研究的方法论，让研究成果能以产品的形式真正面世。

虽然听起来有些戏剧性，但我还是想在介绍第 2 版之前提一提写第 1 版时的个人情况。尽管那一版书对我来说有特殊的意义，但对于数据科学这个欣欣向荣的领域而言只做出了很小的贡献。但是我为它而自豪。我在那本书中获得了救赎，它让我重新找回了感觉，让我及时从病痛中恢复，让我摆脱失败的痛苦而获得了成就的喜悦，这就是第 1 版的情况。

在第 2 版中，我希望能做到更多。简单地说，我希望能引导初出茅庐的数据科学家，让其快速成长为数据分析应用开发者。我把自己在三个 Hadoop 团队与一个 Spark 团队中获得的构建分析应用的经验进行了总结和提炼。这次改版中，编程语言使用的是数据科学的通

用语言 Python，而选择的大数据平台是 Spark。希望本书能成为读者的必备指南，让读者快速学会如何构建足以应对各种数据规模的分析应用。

Spark 取代 Hadoop/MapReduce 成为了处理大规模数据的主流方式，因此我们在这一版中使用 Spark 来讲解。不仅如此，根据我们团队在工作中对敏捷数据科学的进一步理解，本书对敏捷数据科学方法论的理论和发展也做了进一步完善。希望第 1 版的读者还可以从第 2 版中获得提高，也希望比起相对更适合 Hadoop 用户阅读的第 1 版，这一版能更好地服务于 Spark 用户。

敏捷数据科学有两大目标：一是为了使用 Python 和 Spark 搭建出任意规模的数据分析应用，二是帮助产品团队学会使用敏捷的方式协作开发分析应用来保障工作成效。

敏捷数据科学的邮件列表

你可以在邮件列表（*agile-data-science@googlegroups.com*）或网页（*https://groups.google.com/d/forum/agile-data-science*）中学到最新的敏捷数据科学知识。

我为本书维护了一个网页（*http://datasyndrome.com/book*），里面有最新的更新，以及为读者准备的相关资料。

产品分析咨询公司 Data Syndrome

我创办了一家叫作 Data Syndrome 的咨询机构（见图 P-1）来推广本书中的方法论和技术栈。如果你要在你的公司里实践敏捷数据科学并且需要这方面的帮助，或者是需要构建数据产品方面的帮助，又或者需要“大数据”方面的培训，你可以通过我的邮箱（*rjurney@datasyndrome.com*）或网站（*http://llc.datasyndrome.com/*）来联系我。

Data Syndrome 提供视频课程《使用 Kafka、PySpark、Spark MLlib 和 Spark Streaming 进行实时预测分析》（*Realtime Predictive Analytics with Kafka,PySpark, Spark MLlib and Spark Streaming. http://datasyndrome.com/video*），使用了第 7 章和第 8 章的材料，教观看者如何用 Kafka、Spark Streaming 及网络应用的前端页面构建出整套的实时预测系统（见图 P-2）。如果想进一步了解，请访问 *http://datasyndrome.com/video* 或联系 *rjurney@datasyndrome.com*。

图 P-1　Data Syndrome

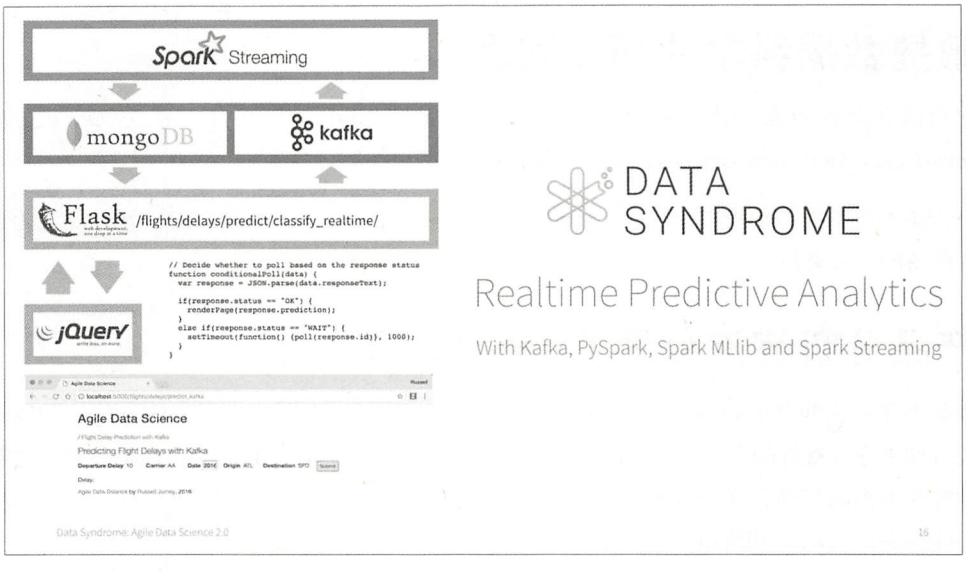


图 P-2　实时预测分析视频课程

在线培训

Data Syndrome 正在研制针对数据科学团队和数据工程团队的全套在线大数据培训课程。目前提供的课程可以根据需求进行自定义，包括以下几个主题。

敏捷数据科学

持续三天的课程，涵盖了全栈分析应用的构建。在内容上与本书相近，可以将数据科学家训练为全栈应用开发者。

实时预测分析

一天即可完成，时长总共 6 小时。包括如何使用 Kafka 和 Spark Streaming 及网络应用前端页面构建整套的实时预测系统。

PySpark 介绍

一天即可完成，时长 3 小时。向参与者介绍如何使用 Spark 的 Python 接口进行基本的数据处理。最终教会参与者如何使用 Spark MLlib 构建一个分类器模型来预测航班延误。

详情请访问 *http://datasyndrome.com/training* 或联系 *rjurney@datasyndrome.com*。

本书目标读者

本书的目的是帮助初学者和初出茅庐的数据科学家成长为数据科学与数据分析团队的主力成员。本书想要帮助工程师、分析师、数据科学家以敏捷的方式来使用 Hadoop 在大数据上进行工作。本书介绍的敏捷方法论很适合大数据领域。

本书是为需要开发软件来分析数据的程序员而写的。设计师和产品经理可能更适合第 1 章、第 2 章和第 5 章，这些章节主要作为敏捷过程的导论，没有专注于编码运行。

本书假设你在类 UNIX 环境中工作，没有为 Windows 用户提供示例，不过 Windows 用户可以使用 Cygwin 尝试。

本书主要结构

本书分为两个部分。第 I 部分介绍的是我们在第 II 部分中需要用到的数据集和工具集。第 I 部分故意写得简明扼要，只是为了尽可能快地介绍这些工具。第 II 部分会更深入地探讨这些工具的使用，所以如果在读第 I 部分时感觉有些不知所措也不用担心。第 I 部分的章节如下。

第 1 章 理论

介绍敏捷数据科学的方法论。

第 2 章 敏捷工具

介绍要用的工具集，并且讲解工具如何上手与安装。

第 3 章 数据

描述本书中使用的数据集。

仅供非商业用途或交流学习使用

第Ⅱ部分是我们使用敏捷数据科学来构建一个分析应用的教程。这是一份笔记本式的分析应用构建指南。我们逐层攀登数据价值金字塔，始终应用敏捷的原则。这一部分会展示在敏捷迭代进程中一步一步发掘数据价值的方法。第Ⅱ部分由以下所列章节组成。

第 4 章 记录收集与展示

帮你下载航班数据，并且通过网络应用展示航班记录。

第 5 章 使用图表进行数据可视化

一步步引导你如何在网络应用中加入一些简单的图表来展示数据。

第 6 章 通过报表探索数据

教你如何从数据中提取出实体关系，将其参数化并相互关联以创建交互式的报表。

第 7 章 进行预测

在先前所做的基础上对某一航班准点与否进行预测。

第 8 章 部署预测系统

展示如何部署预测系统来确保真正发挥作用。

第 9 章 改进预测结果

不断迭代提高我们的准点航班预测应用的表现。

附录 A 安装手册

展示如何安装所需工具。

本书样式约定

本书使用以下所列样式约定。

斜体（*Italic*）

表示新术语、URL、电子邮箱地址、文件名、文件扩展。

等宽字体（`Constant width`）

用于程序示例，还有在文字中引用的程序中的内容，比如变量名或函数名、数据库、数据类型、环境变量、语句，还有关键字。

加粗等宽字体（**`Constant width bold`**）

表示需要用户按字面输入的命令或其他文本。

等宽斜体（*`Constant width italic`*）

表示需要以用户提供的值或上下文决定的值替换的文本。

仅供非商业用途或交流学习使用

本图标表示一个提示、建议或注释。

本图标表示一个警告或提醒。

代码示例的使用

补充材料(代码示例、练习等)可以在 *https://github.com/rjurney/Agile_Data_Code_2* 中下载到。

本书是要帮你解决问题的。总的来说，你可以在你的程序或文档中直接使用本书所提供的示例代码。除非要大部分照搬代码，否则你不需要联系我们获取许可。举例来说，写一个程序时从本书中抄几段代码片段不需要许可。出售或分发 O'Reilly 书籍代码示例的光碟也不需要许可。在回答问题时引用本书并且摘抄示例代码不需要许可。但是如果要在你的产品文档中大量使用示例代码，你就需要申请许可。

我们鼓励但是不要求标明出处。通常情况下，引用要包含题目、作者、出版商、ISBN 码等信息。你可以这样标明对本书的引用："*Agile Data Science 2.0* by Russell Jurney (O'Reilly). Copyright 2017 Data Syndrome LLC, 978-1-491-96011-0."

如果你觉得你对示例代码的使用超出了正常的范围或上面列出的许可范围，请通过 *permissions@oreilly.com* 联系我们。

O'Reilly Safari

Safari (前身为 Safari Books Online) 是一个会员制的培训与参考平台，为企业、政府、教育机构和个人提供服务。

会员可以访问数以千计的书籍、培训视频、学习路径、交互式指南、编排的播放列表等来自超过 250 家出版商的资料。这些出版商包括 O'Reilly Media、Harvard Business Review、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Adobe、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 及其他一些出版商。

详情请访问 *http://oreilly.com/safari*。

仅供非商业用途或交流学习使用

如何联系我们

关于本书的意见和建议，可以通过以下方式联系出版商：

美国：
O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：
北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

对于本书的评论或技术性问题，你可以发邮件到 *bookquestions@oreilly.com*。

关于我们的书籍、课程、会议、新闻的更多信息，请访问我们的网站 *http://www.oreilly.com*。

在 Facebook 上找到我们：*http://facebook.com/oreilly*。

在 Twitter 上关注我们：*http://twitter.com/oreillymedia*。

在 YouTube 上观看我们：*http://www.youtube.com/oreillymedia*。

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **下载资源**：本书如提供示例代码及资源文件，均可在下载资源处下载。
- **提交勘误**：您对书中内容的修改意见可在提交勘误处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方读者评论处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/35166

仅供非商业用途或交流学习使用

第 I 部分

准备工作

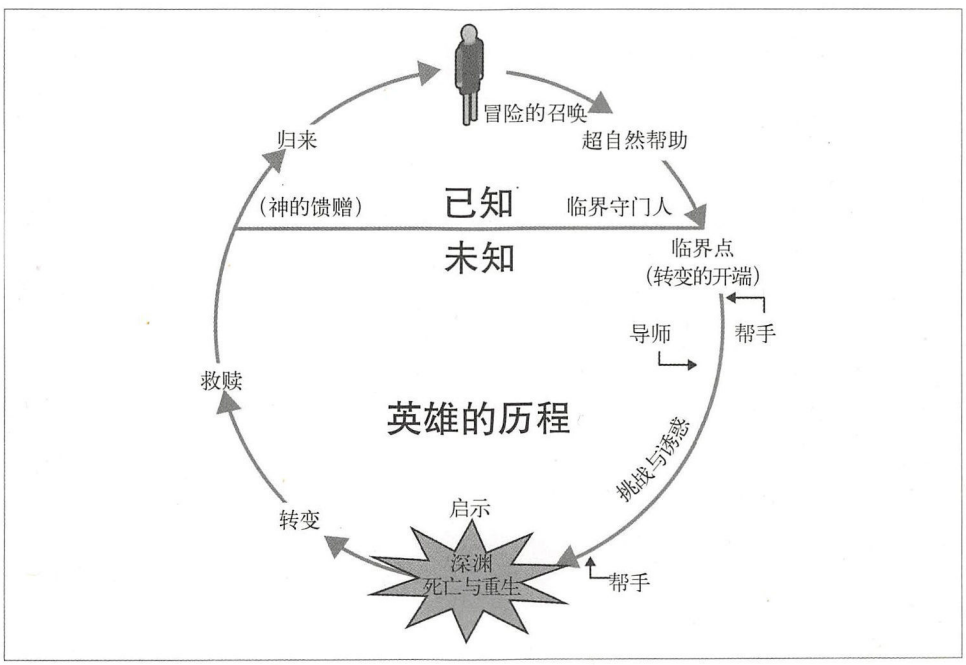


图 I–1　英雄的历程，来自维基百科（*https://en.wikipedia.org/wiki/Hero%27s_journey*）

仅供非商业用途或交流学习使用

第 1 章

理论

我们一直在实践中探寻更好的软件开发方法，同时也身体力行地帮助他人。由此我们建立了如下价值观：

个体和互动高于流程和工具；
工作的软件高于详尽的文档；
客户合作高于合同谈判；
响应变化高于遵循计划。

也就是说，尽管右项有价值，但我们更重视左项的价值。

——敏捷软件开发宣言（*http://agilemanifesto.org*）[1]

导论

敏捷数据科学（*Agile Data Science*）是一种围绕网络应用开发展开的数据科学实践。敏捷数据科学认为，适用于在机构中发挥作用的数据科学过程的最有效的输出应为网络应用程序。它还认为应用开发是数据科学家的基本技能。因此，从事数据科学变成了构建用于描述应用研究过程的应用程序，包括快速的原型设计、探索性数据分析、交互式可视化以及应用机器学习。

敏捷开发已经成为当今软件交付的实际方式。敏捷开发包括一系列成熟的方法学，比如

1 著作权为以下作者所有：Kent Beck、Mike Beedle、Arie van Bennekum、Alistair Cockburn、Ward Cunningham、Martin Fowler、James Grenning、Andrew Hunt、Ron Jeffries、Jon Kern、Brian Marick、Robert C. Martin、Steve Mellor、Ken Schwaber、Jeff Sutherland、Dave Thomas。2001 年此宣言可以以任何形式自由地复制，但其全文必须包含上述申明。——译者注

仅供非商业用途或交流学习使用

Scrum，这些方法学提供了以小增量构建出优秀软件的框架。已经有人尝试过将软件的敏捷开发应用到数据科学中，但是结果并不理想。对于敏捷过程的产物而言，交付用于生产环境的软件和影响决策的认知是截然不同的。由于我们需要认知来影响决策，因此数据科学有了不确定性因素。对于软件来说，我们有“完成”的概念，但是软件还是缺了些什么，毕竟软件不能产出真实的、影响决策的认知。正如数据科学家 Daniel Tunkelang 所说：“影响决策的认知必然比软件工程松散。”Scrum 和其他一些敏捷开发方法学并没有很好地处理这种不确定性。简单来说，敏捷数据科学不能直接照搬软件的敏捷开发。这就是本书创作的动机：提供一种适合数据科学的不确定性的新方法学，并指导如何将其应用到实践中，我们会通过实际的软件开发来展示其中的一些基本原则。

为了将敏捷开发经验应用到数据科学实践中，我尝试提出了敏捷数据科学“宣言”这样一套严谨的方法。这些原则不仅仅适用于数据科学家在生产中构建数据产品。网络应用程序是在机构内和机构间共享能影响决策的认知的最佳形式。

敏捷数据科学不仅仅包括交付能用的软件，还包括更好地把数据科学整合到机构内其他团队的开发中。数据科学与工程之间存在长期的错位，数据科学团队在进行探索性的数据分析和应用研究，工程团队却经常不理解数据科学团队在做什么。同时，工程团队也常常不确定该做些什么，这样就造成了“瀑布的拉力”，敏捷开发的项目似乎也呈现出了瀑布模型的特征。敏捷数据科学把这两种团队连接到了一起，拧成了一股绳。

这本书也是关于“大数据”的。敏捷数据科学是一种开发方法学，可以应对基于大规模数据构建数据分析应用的各种不可预知的情况。它是操作 Spark 进行数据提炼的理论与技术指南，让我们能在公司内充分发挥“大数据”的力量。数据仓库级别的运算给了我们巨大的存储空间和丰富的计算资源，现在我们可以解决需要对史无前例的数据量进行存储和处理的新问题了。我们的新工具包括更强大的处理器、更大容量的存储，以及虚拟化技术、统计学、机器学习技术。如今，通过引入这些新工具，我们可以解决一些以前近乎无解的难题；还可以从原始数据中获得全新的产品，把原始数据提炼为可以转化为利润的认知，在新型的数据分析应用中将这些认知产品化并且应用到生产环境中，这都使我们产生了浓厚的兴趣。这就是数据科学（*data science*）。

与此同时，在过去的 20 年中，万维网已经成为了最主要的信息交流媒介。在这段时间中，软件工程领域也发生了“敏捷”革命，应用的构思、构建、维护都与之前大不一样。这些新现象使得许多项目和产品能够按期交付、不超预算，也让小团队和个人开发者能在各行各业中开发出完整的应用。这就是敏捷软件开发（*agile software development*）。

但是问题也随之而来。操作真实而原始的数据、从事数据科学、进行严肃的研究，这些都

仅供非商业用途或交流学习使用

很花时间，比敏捷开发的周期（一般是“月”这个量级）要长得多。许多机构的开发迭代周期并没有那么长，也就是说如今应用研究者的进度压力相当大。数据科学受困于例如瀑布模型之类的老式软件工程开发模式的桎梏。

在数据科学和敏捷开发这两股潮流的交汇处，我们的问题和机会都出现了：数据科学属于应用研究，需要在无法给出确切时间计划的情况下竭尽全力，我们如何才能将它和应用的敏捷开发结合起来？数据分析应用的开发管理如何能超越早已被我们抛弃的瀑布模型？我们要如何为未知的、不断演进的数据模型制作应用程序？我们要如何开发新的敏捷开发模型来适应数据科学过程，创造出伟大的产品？

本书尝试整合敏捷开发和大数据集上的数据科学这两个领域；同时将研究和工程融合到一起，产出一些有用的产品。为了实现这一点，本书提出了一套新的敏捷方法学，并且提供了一些使用合适的软件栈构建产品的例子。这套方法学旨在基于最透彻的认知，最大限度地提升软件功能。而本书的软件栈则是一套轻量级的工具集，足以应对不确定且不断变化的原始数据，并提供足够的生产力驱动敏捷开发过程的成功实现。本书会继续展示如何使用这套软件栈迭代地创建价值，重新获得敏捷性，从数据中挖掘出价值并转化成真金白银。

敏捷数据科学的目的是让你能掌控大局，确保你的应用研究能产出有用的产品，满足实际用户的需求。

定义

什么是敏捷数据科学（Agile Data Science，ADS）？在本章中，我提出了一种新的开发分析产品的方法学，我在第 1 版中曾暗示过但是没有明确地表达出来。我们先从敏捷数据科学过程的目标开始说。

方法学

敏捷数据科学的目标是记录探索性数据分析的过程并促进和指导这一过程，以期发现实现一款引人注目的数据分析产品的关键路径，并沿着这条路走下去（见图 1-1）。敏捷数据科学深入本质，关注探索性数据分析的过程，并记录在过程中获得的认知。敏捷数据分析把这些当作产品开发的主要工作。通过抓住本质，我们把整个过程的焦点放在可预测的事物上，而不是放在产品的不可预测的输出上，这样便于我们管理整个过程。

仅供非商业用途或交流学习使用

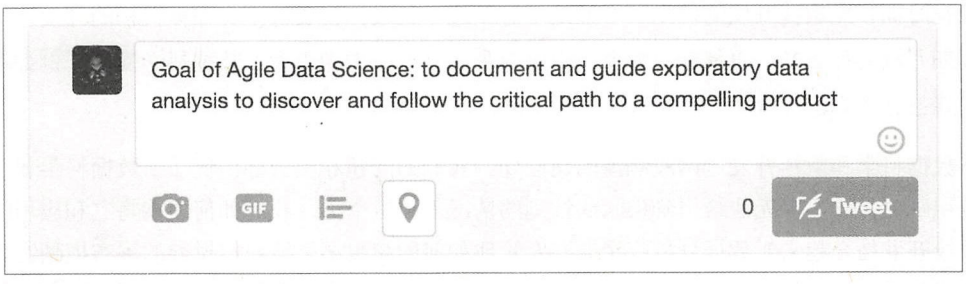


图 1-1　以推文形式显示的方法学

针对数据科学，我们需要一套全新的敏捷宣言。

敏捷数据科学宣言

敏捷数据科学是围绕下面这几条原则进行组织的。

- 迭代，迭代，再迭代：表格、图表、报表、预测。
- 发布阶段性结果。即使是失败的实验也有输出。
- 原型实验高于实施任务。
- 在产品管理中仔细考虑数据的情况。
- 不断上下求索于数据价值金字塔。
- 找到实现杀手级产品的关键路径并遵照执行。
- 抓住本质。描述过程而不仅仅是最终状态。

让我们依次详细阐述这些基本原则。

迭代，迭代，再迭代

见解来自一系列查询中的第二十五条查询，而不是第一条查询。数据表必须被解析、格式化、排序、汇总和总结，然后才能被理解。有见解的图表通常来自第三次或第四次尝试，而不是第一次。构建准确的预测模型可能需要经过多轮特征工程和超参数调优。在数据科学中，迭代是提取、可视化以及产品化见解不可或缺的环节。构建数据科学应用就意味着迭代。

发布阶段性结果

迭代是构建分析型应用不可或缺的行为，这意味着每个冲刺结束时我们都会留下一些没做完的事情。如果我们不在冲刺结束时发布未完成的阶段性输出，可能最终什么也没发布出去。那不是敏捷开发该有的样子；我把它称为“死亡循环”，浪费了无数的时间打造出的

东西却压根没人要。

好的系统是自我记录的，而在敏捷数据科学中，我们记录和分享我们在工作中创建的未完成的数据。我们把所有工作都提交到代码管理系统中。我们与团队成员分享的工作成果，也应尽快与最终用户分享。这一原则并不是对每个人来说都是那么容易接受的。许多数据科学家有着学术背景，他们曾经经过多年的紧张研究，仅仅写出一纸毕业论文，用作申请学位。

原型实验高于实施任务

在软件工程中，产品经理让开发者在一个冲刺阶段内完成一张图表的开发。开发者把这个任务转化为实现一个 SQL GROUP BY 语句，并创建一个网页来显示图表。任务完成了吗？没有。如此具体的图表不大可能有什么价值。数据科学和传统软件工程的不同之处就在于它一半是科学一半是工程。

对于任何任务，我们都需要迭代以获取见解，我们可以把这些迭代当作实验。管理数据科学团队意味着监督多个并行开展的实验，而不仅仅是分配任务。良好的数据资产（表格、图表、报表、预测）是作为探索性数据分析的结果出现的，所以我们应该多从实验的立场思考问题而不是完成任务的立场。

在产品管理中仔细考虑数据的情况

能做成什么与想做成什么一样重要。知道孰难孰易与知道需要什么一样重要。在软件开发中，有三个方面需要考虑：客户、开发人员、业务。而在分析型应用程序开发中，还有一个方面需要考虑：数据。如果不掌握数据对于某个功能“不得不说的意见”，产品负责人就不算尽职。在产品讨论中要始终考虑数据的情况，这意味着产品讨论必须以可视化为基础，我们的工作重点在于通过内部应用程序进行探索性数据分析。

上下求索于数据价值金字塔

数据价值金字塔（见图 1-2）是模仿马斯洛需求层次理论发展出的五层金字塔模型。它表达了原始数据经过表格、图表、报表到预测模型的层层提炼所包含的越来越多的价值，每一层的目的都是提供更多行为依据或是改进现有的行为：

- 数据价值金字塔的第一层（记录）与连通（*plumbing*）有关，目的在于打通数据集从采集到在应用中展示之间的流程。

- 图表（*chart*）和表格（*table*）层是数据优化和分析的起点。
- 报表（*report*）层提供沉浸式数据探索，帮助推理并了解数据。
- 预测（*prediction*）层创造了更多价值，但要创建好的预测模型，就需要进行特征工程，这需要低层的支持。
- 最后一层，行动（*action*）层，它是人工智能热潮发挥作用的一层。如果你所得出的见解无法产生新的行动或是改进现有行动，那这个见解就不那么有价值。

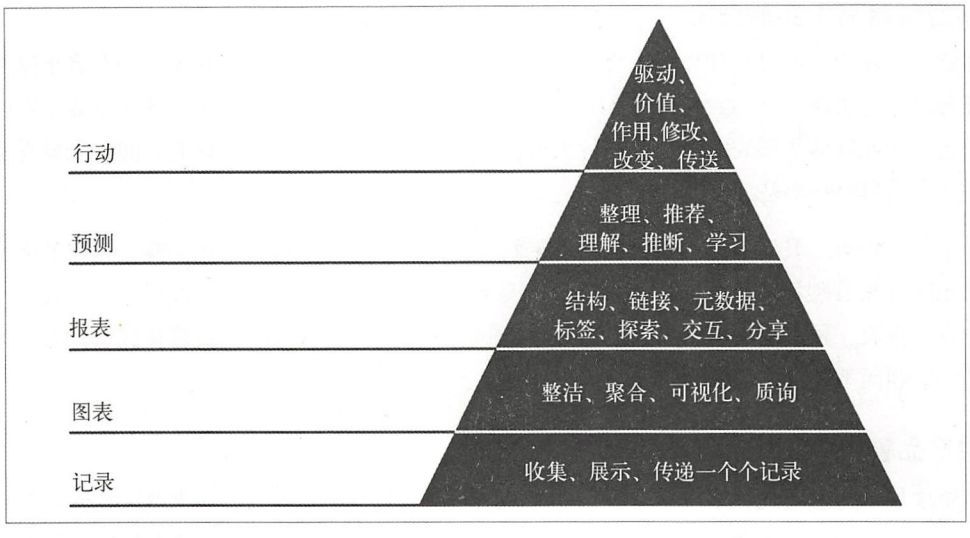


图 1-2　数据价值金字塔

数据价值金字塔为我们的工作制订了提纲。我们应该不仅仅要把这个金字塔当作要遵守的规则，还要时刻牢记在心。有的时候你会跳过一些步骤，有的时候你会搞错顺序。如果直接把新数据放到预测模型中，而且不在底层把数据添加到应用的数据模型中使得数据集可以访问且对于应用透明，那么技术债就来了。你需要牢记，应该尽可能减少技术债。

找到实现杀手级产品的关键路径并遵照执行

为了尽可能地提高我们成功的概率，我们应该把大多数时间放在应用程序对于成功最重要的部分上。但是最重要的是哪部分呢？这需要通过实验才知道。分析型产品的开发就是搜寻和追求移动的目标。

一旦确定了一个目标，比如要做出预测，我们就必须找到实现目标的关键路径（https://en.wikipedia.org/wiki/Critical_path_method）。如果这个目标被证明是有价值的，我们还要找

仅供非商业用途或交流学习使用

到改进的关键路径。随着一个又一个任务的处理，数据被逐步提炼。分析型产品常常需要多个阶段的提炼，涉及 ETL（提取、转化、加载）过程、统计学技术、信息访问、机器学习、人工智能、图分析等。

这些阶段的互动可以形成复杂的依赖网。团队领导者应当掌握依赖网。他的工作就是确保团队发现关键路径，然后组织团队按关键路径完成。产品经理无法自上而下管理关键路径，相反这应当由产品科学家自下而上找到。

抓住本质

如果我们无法轻易按照开发常规应用的日程交付做好产品，那我们交付什么呢？如果不交付，我们就没有做到敏捷。为了解决这个问题，在敏捷数据科学中，我们“抓住本质”。核心就是记录分析过程，而不是我们要开发的应用的终极状态。在迭代攀登数据价值金字塔的过程中交付阶段性内容，遵循杀手级产品的关键路径，这让我们能够敏捷。所以，产品到底从何而来？从记录探索性数据分析的过程而来。

小结

这七条原则共同驱动了敏捷数据科学方法学。它们为探索性数据分析，并且据此为构建分析型应用程序提供了框架和参考。可以说这是敏捷数据科学的核心。但是为何要使用这种方法呢？我们是怎么得出这种方法学的呢？让我们看一个瀑布模型的项目来理解这种项目的问题。

领英职业探索器（Career Explorer）是 2010 年领英使用瀑布模型开发的一款分析型应用程序，它最终的失败是本书写作的动机。我曾被聘为职业探索器的高级数据科学家。在本书中，我把职业探索器作为案例，简要分析八个月的开发过程中瀑布模型暴露出来的问题。

仅供非商业用途或交流学习使用

瀑布模型的问题

我应该解释一下，事实上职业探索器是我构建的第一个推荐系统或者预测模型。它失败的很大原因在于我没有经验。当时我的经验主要在迭代开发和敏捷的交互式可视化上，这其实很符合项目的目标，然而实际的推荐任务比在原型阶段预想的要困难得多。事实证明，职位头衔实体解析的工作量比预估的要大得多。

与此同时，由于项目所采用的方法学的问题，产品的真实状态在管理人员面前隐藏了，导致管理人员直到发布前几天还看着静态模型很开心。最后一刻的整合才揭示了暴露给客户的组件之间接口上的错误。当我们发现产品无法交付时，距离硬性截止时间只剩短短几天，这给我们造成了危机。最后，我差不多熬了一个星期的夜，每五分钟重新提交一次 Hadoop 作业来调试最后一刻的修复和修改，产品才勉强搞定了。事实证明这并不重要，因为用户对这个产品概念并不感兴趣。最后，在发布之后仅仅几个月，大量的工作就被抛弃了。

这个项目的主要问题就是使用了瀑布模型开发。

- 应用概念（*application concept*）只在用户关注点小组和管理层审核中验证过，没有真正考虑用户的想法。
- 预测的表现形式（*prediction presentation*）已经提前设计好了，而真正的预测模型之后才开始构建。事情就变成了这样：
  “我们做的产品设计得很好！你的工作是预测未来。”
  “什么导致对未来进行可靠预测需要这么久？”
  “用户不理解什么叫 86% 正确。”
  飞机→撞山。
- 图表（*chart*）是由产品设计定义的，无法真正提供见解。
- 与客户签订的合同明确了不可宽限的截止日期（*hard deadline*）。
- 整合（*integration*）测试在开发结束时才进行，这会导致截止时间前的危机。
- 为了让各团队专注自己的工作同时便于管理，没有真实数据的模型（*mock-up*）在项目中随处可见。

仅供非商业用途或交流学习使用

这都是瀑布模型中相当标准的做法。结果就是管理人员一直认为产品在按计划推进，直到还剩两周时间时，在整合阶段终于发现了问题。注意我们整个项目过程中始终使用 Scrum 模式，然而最终产品完全经不起终端用户的测试，因此我否认用了敏捷开发的模式。总结起来就是，飞机撞山上了。

与之相反，在领英我还主持开发和管理了另一个叫作 InMaps（https://techcrunch.com/2014/09/01/linkedin-is-quietly-retiring-network-visualization-tool-inmaps/）的项目。这个项目的进展要顺利得多，因为我们使用真实数据迭代发布应用，把“坏”的状态暴露给内部用户，在许多发布周期中收集反馈。这两个项目截然相反的经历促使我形成了敏捷数据科学的想法。

但是如果在职业探索器项目中实际使用的方法是 Scrum，那么为什么说它是瀑布模型的项目呢？因为数据科学团队构建的分析型产品会被一种神秘的力量“拉”向瀑布模型。稍后我会解释这种趋势的成因。

研究与应用开发

发布分析产品的基本矛盾就是研究与应用开发时间线之间的矛盾。这样的矛盾容易把所有的分析产品都变成瀑布模型的项目，尽管有些项目从一开始就是使用像 Scrum 这样的软件工程方法学进行开发管理的。

研究，哪怕是应用研究，都是科学。它涉及迭代实验，也就是一个实验的结果影响下一个实验的设计。科学擅长发现未知，与工程相比的不同之处在于，科学没有特定的终点（见图 1-3）。

仅供非商业用途或交流学习使用

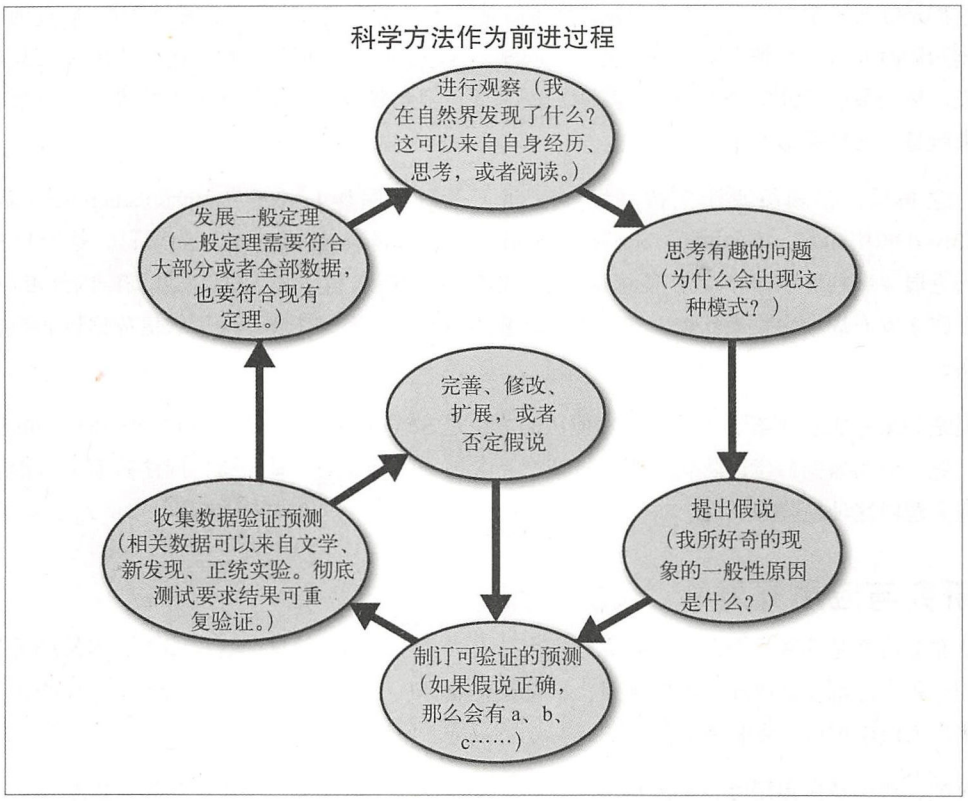


图 1–3　科学方法，来自维基百科（*https://en.wikipedia.org/wiki/Scientific_method*）

工程是运用已知的科学知识和工程技术，以线性的流程来构建事物。如图 1-4 中的甘特图所示。任务可以被细化、监控和完成。

仅供非商业用途或交流学习使用

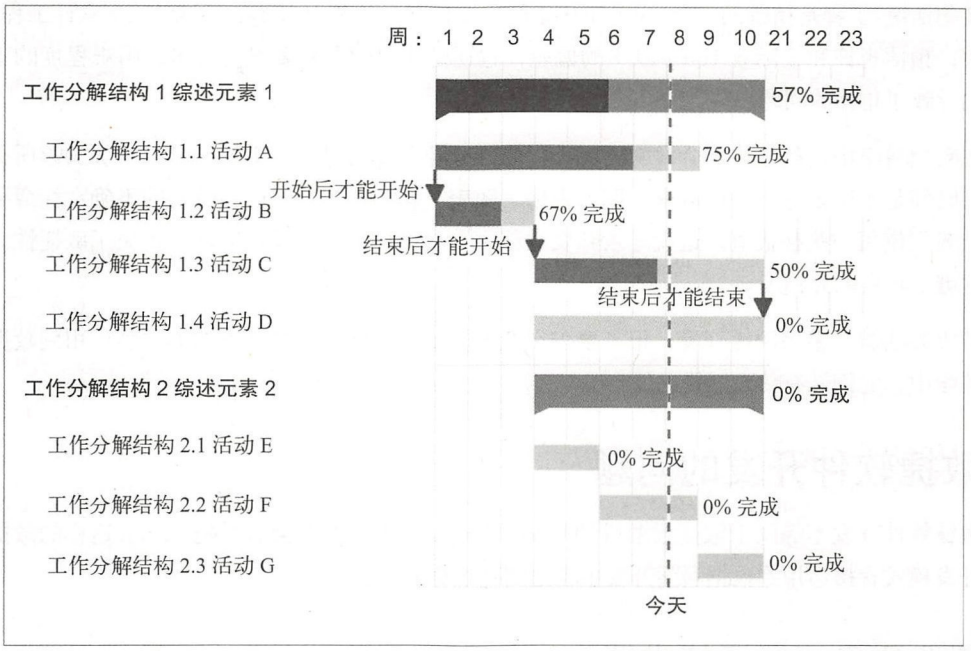


图 1–4　甘特图，来自维基百科（*https://en.wikipedia.org/wiki/Gantt_chart*）

工程项目中的一种更好的模型如图 1-5 中的 PERT 图所示，可以建立复杂的非线性依赖关系。注意即使在这个更高级的模型中，那些点还是未知的。线表示工作。

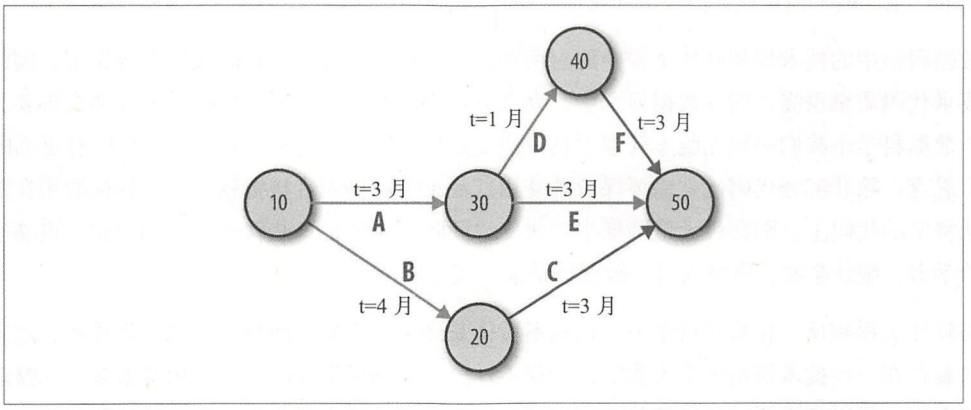


图 1–5　PERT 图，来自维基百科（*https://en.wikipedia.org/wiki/Program_evaluation_and_review_technique*）

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

换句话说：工程是精确的，而科学是不确定的。即使在一些相对较新的领域（比如软件工程）中，预估也许和实际有 100% 以上的偏差，也还是比科学过程要确定得多。困难程度的不同导致了这样的问题。

在数据科学中，科学的部分常常要比工程的部分花更多的时间。更糟的是，一次实验所花的时间是不确定的。分析成果（例如表格、图表、预测模型等）产出时间的不确定性会导致我们使用一些替代值。这就导致开发过程由来自模拟数据的反馈驱动，丧失了敏捷性。这就是项目的死路。

解决办法就是采用敏捷开发，但是要怎么做呢？如何把软件敏捷开发的方法学应用到数据科学中，它们又有什么不足之处？

敏捷软件开发的问题

敏捷软件开发不等同于敏捷数据科学开发。在这一节中，我们会看看把 Scrum 这样的敏捷开发模式直接运用到数据科学开发中会遇到什么样的问题。

最终质量：偿还技术债

技术债（*https://www.techopedia.com/definition/27913/technical-debt*）被 Techopedia 定义为："编程中的一个概念，指因为使用了容易实现的代码而不是全局最佳解决方案的代码，导致的额外开发工作。"理解技术债是管理软件应用开发的关键，因为截止时间的压力会导致大量技术债的产生。技术债可能会严重破坏将来截止时间前赶工的能力。

数据科学中的技术债和软件工程中的有所不同。在软件工程中所有的代码都要保留，因此要求代码质量很高。而在数据科学中，大部分代码都要丢掉，所以代码质量没那么重要。在数据科学中我们必须在版本管理工具中提交全部代码，同时必须对丑陋的代码有更高的容忍度，除非部分代码确定需要保留并复用。如果不这样而直接把软件工程的标准用在数据科学的代码上，则会极大地降低生产效率。同时，通过把一些软件工程知识和习惯灌输给学者、统计学家、研究人员、数据科学家，代码质量可以提高很多。

与软件工程相反，在数据科学中，代码不应该*始终是优质的*，而只要*最终是优质的*。这意味着存在一些技术债是无伤大雅的，只要不过分。要确保重要的代码不用费太多力气就能清理好。代码不需要时刻保持整洁，但是一旦代码成为了重要代码，就必须是整洁的。技术债组成了管理敏捷数据科学项目的依赖关系网的一部分。技术债的偿还是一项技术性很强的任务，要求团队领导者具有技术技能，或者干脆由团队中其他成员来实现。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

原型是由技术债支撑的，只有在原型被确定为有用的时候，技术债才需要偿还。大多数原型会被抛弃或者极少用到，因此其中的技术债不会产生影响。这可以让我们利用有限的资源实现更多的实验。这在 Jupyter 和 Zeppelin 中也有体现，它们强调直接的表达，而不是代码复用或者生产环境部署。

瀑布模型的拉力

现代“大数据”应用的栈比起传统应用程序要复杂得多。而且，使用这些系统构建大规模分析型应用需要丰富的技术能力。人员和技术的多样性会导致项目被一种力量“拉”向瀑布模型，即使团队希望能采用敏捷开发。

图 1-6 展示了如果任务在一个个冲刺中完成，笨重的技术栈和团队会使项目回到瀑布模型。在这种情况下要画一个图表，于是数据科学家用 Spark 计算出图表的数据并把数据存入数据库。接下来，API 开发人员为这份数据设计了 API，然后网站开发人员为图表开发了一个网页。可视化工程师创建了真正的图表，然后设计师对图表进行了美化。最后，产品经理看到了图表，之后如果要修改的话又是一轮新的迭代。每前进一步都要多花很多时间。进度非常缓慢，团队也谈不上敏捷。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

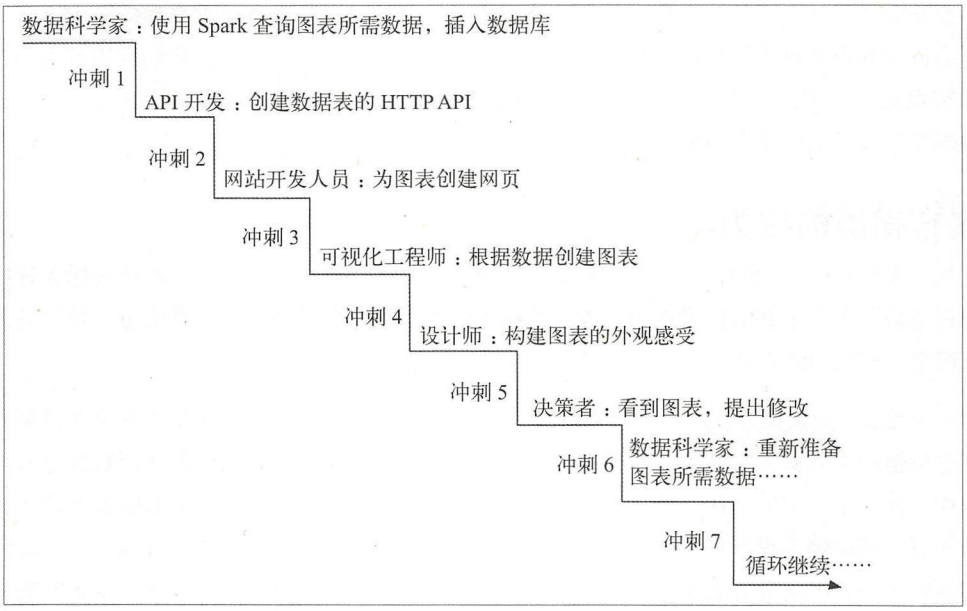


图 1-6　基于多轮冲刺的协作最终与敏捷无关

这说明了几件事。首先就是能完成好几个相关任务的通才很有用。更重要的是，我们有必要在冲刺内部进行迭代，而不是以冲刺为周期进行迭代。否则，如果冲刺中都在等一个团队成员完成前一个成员的工作，整个过程就像一级级瀑布一样了。

数据科学过程

介绍了方法学以及方法学的作用之后，让我们深入敏捷数据科学团队的运作机制。我们从设置预期开始，然后了解数据科学团队中的各种角色，最后讲解数据科学过程在实践中怎样工作。尽管我希望这能为新接触数据科学团队或者敏捷数据科学的读者提供介绍，但这还不是敏捷过程整个运作方式的完全讲解。在读本章之前，我建议不了解敏捷开发或者数据科学的读者先找一本关于 Scrum 的书读一读。

现在让我们谈谈如何设置数据科学团队的预期，以及如何与机构中其他的团队互动。

仅供非商业用途或交流学习使用

设置预期

在我们讨论如何组建数据科学团队并让团队运作产出可行动的见解之前，我们首先需要讨论数据科学团队如何在机构中运作。由于敏捷数据科学的关注点从预定义的输出转向了应用研究过程的描述，因此敏捷数据科学团队的预期也要相应改变。另外，数据科学团队与其他团队的协作关系也要相应调整。

“我们什么时候交付？”这是管理人员想了解的问题，他们需要知道问题的答案来为客户设置预期，协调销售、市场、招聘等各方面工作。对于敏捷数据科学团队来说，这个问题没有非常明确的答案。无法确定特定的发布日期 X，也无法确定网络应用或者 API 所能提供的预测模型 Y。具体的交付日期和交付产品的内容都是无法保证的，这也是采用敏捷数据科学过程所牺牲的。而所获得的则是真正看到团队向业务目标前进的进度，因为可以时刻看到描述团队做了些什么的软件。有了这一信息，其他的业务流程也可以与数据科学的实际情况相匹配，而不是按照臆想的发布日期和产品安排工作。

目标不明确就引出了另一个同样重要的问题：“我们交付什么？”或者更可能这么问：“我们将来交付什么，什么时候交付？”为了回答这些问题，决策者可以查看当前应用的情况，以及下一个冲刺的计划，以了解当前的情况如何，以及下一步会去向何方。

解决了这两个问题，整个机构就可以配合数据科学团队工作了，而数据科学团队的工作也逐渐进化为可触发行动的见解。数据科学团队的任务应该是发掘数据的价值，解决业务中的各种问题。他们工作产出采用的形式是通过探索性的研究发现的。“最终”产品的完成日期可以通过仔细审视当前工作进度进行预估。这种信息尽管比发布日期要琐碎很多，数据科学团队周边的经理们还是可以根据它来同步工作进度与日程安排。

换句话说，我们无法告诉你我们到底要在何时交付什么样的东西。不过作为接受了这个现实的补偿，你可以有一个持续交付的进度报告，可以让你根据数据科学研究的实际进展协调其他方面的工作。这是敏捷数据科学的取舍。由于提前定义好产出物和产出时间的日程通常定义了错误的产出物和不现实的时间，所以我们认为这是一种明智的取舍。事实上，这也是我们面对数据科学的实际情况唯一能做的。

仅供非商业用途或交流学习使用

数据科学团队的角色

产品是由一组成员共同构建的，因此敏捷开发的方法学更关注人员而不是过程。而数据科学是一个宽泛的概念，横跨分析、设计、开发、业务、研究等领域。敏捷数据科学的团队成员也因此包含从客户到运维的各种角色，见图 1-7。

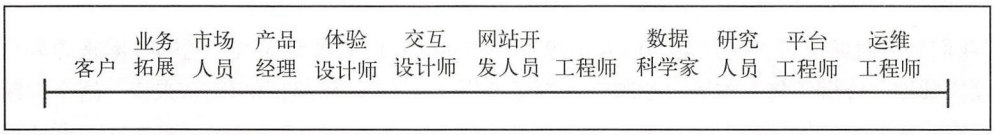


图 1-7　敏捷数据科学团队成员的角色

这些角色的具体定义如下。

- *客户*：产品的使用者，单击你提供的按钮和链接，或者完全无视。你的任务是为他们不断创造价值。他们是否感兴趣决定了你的产品是否成功。
- *业务拓展*：与早期客户签约，可能是通过面对面联系，也有可能是通过推广页面和活动获取，在市场上为产品造势。
- *市场人员*：与客户直接交流，决定投放何种市场。他们决定了敏捷数据科学产品的出发点。
- *产品经理*：聆听各方观点并进行汇总，构建对于产品愿景的共识，明确产品的发展方向。
- *用户体验设计师*：负责从客户的角度协调设计与数据的关系。这一角色非常关键，因为统计学模型的输出对于对模型输出结构毫无概念的“普通”用户来说，是非常难以理解的（比如什么叫 75% 正确）。
- *交互设计师*：设计与数据模型的交互方式，以便用户发现价值。
- *网站开发人员*：开发把数据显示在浏览器中的网络应用程序上。
- *工程师*：构建为应用程序传递数据的系统。
- *数据科学家*：以创新的方式探索并转化数据以实现并发布新功能，整合各种来源的数据以创造价值。他们和研究人员、工程师、网站开发人员、设计师一起实现数据可视化，不断地及时展示数据，不论是原始数据、阶段性数据还是经过提炼的数据。
- *应用研究人员*：解决数据科学家发现的阻碍发掘价值的严重问题。这类问题需要投入精力与时间，要用统计学和机器学习中的新方法来解决。

仅供非商业用途或交流学习使用

- 平台工程师 / 数据工程师：解决分布式基础架构中出现的问题，让敏捷数据科学可以无痛伸缩。平台工程师负责随时解决严重问题的工单，并根据长期计划实现一些项目来为研究人员、数据科学家以及工程师提供基础平台维护，提高系统可用性。
- 质量保障工程师：对预测系统进行端到端的自动化测试，确保能做出精确可靠的预测。
- 运维 / 开发运维工程师：保障生产环境的数据基础架构的平顺安装和运行。他们要实现自动化部署，出故障的时候会大费周章。

认清机遇与挑战

构建数据产品所需的技能种类繁多，因而蕴含着机遇与挑战。如果一个团队中有技能覆盖全面的专家团队，并且这些专家能在各自的领域中充分运用自己的技能，那么问题就可以很容易地分解，并且各个击破。这样一来，数据科学过程就成了一条高效的装配线，见图 1-8。

仅供非商业用途或交流学习使用

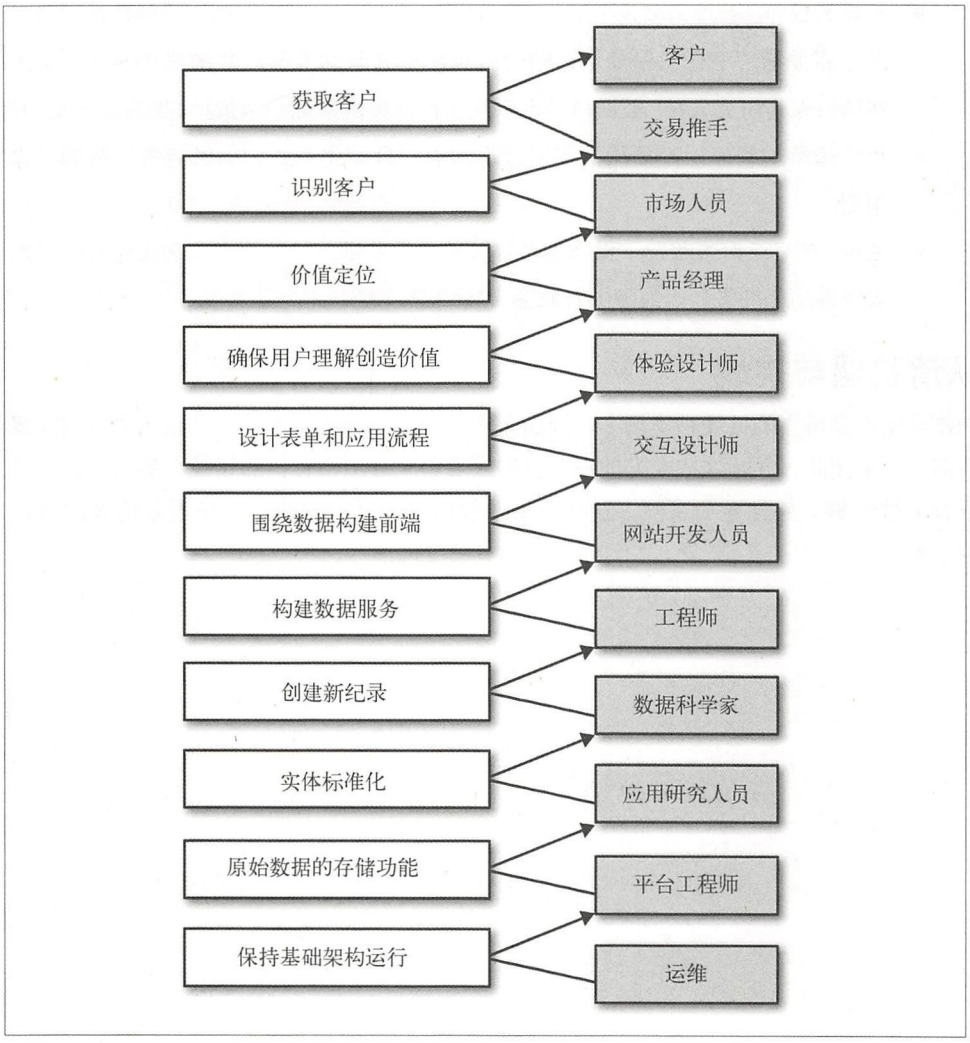


图 1-8　专家各尽其职的工作过程

然而，由于数据科学包含很多不同领域，所需的专业技能也种类繁多，为了适应需求招募各领域的专家，团队规模越来越大，沟通的代价很快会成为最大痛点。如果研究人员与最终客户之间的沟通需要经过八个人，这会让研究人员很难解决真正有意义的问题，相反更可能做无用功。同样地，团队会议如果要十几个人一起参加，就会变得几乎没有效率可言。如果我们把这样的团队分割成一些小的部门，并且约定如何在部门间分工，那我们又会失去敏捷性与整体性。我们需要为研究工作的成果制定一系列标准，并等待研究工作的产出达到这些标准，很快我们就会发现又回到了瀑布模型的老路上。

仅供非商业用途或交流学习使用

我们知道成功构建产品有几个必要条件，包括敏捷性、统一的大局观，以及对产品的共识。产品开发中最糟糕的情况就是团队没有统一的大局观。我们怎样才能整合横跨众多领域的技能，协调应用研究、数据科学软件开发、设计等角色之间脱节的时间计划呢？

适应变化

为了保持敏捷性，我们必须拥抱并适应新环境。我们必须按照精干的方法学适应变化，保持高效的产出。

尤其是以下的一些改变，还会进一步提高我们的敏捷性。

- 优先选择通才而非专才。
- 小规模团队比大规模团队好。
- 使用高级的工具和平台：云计算、分布式系统、平台即服务（PaaS）。
- 将阶段性结果持续迭代输出，即使工作尚未完成。

在敏捷数据科学中，一个由通才组成的小型团队使用可伸缩的高级工具和平台，将数据不断迭代提炼为价值更高的状态。我们积极使用云计算、分布式系统和平台即服务（PaaS）等技术组成的软件栈。即使是最深入的研究，我们也使用这个软件栈对阶段性结果进行迭代发布，以滚雪球的方式，从简单的记录开始，到可以创造价值的预测和行为，让我们在此过程中抓住一些价值，把数据转化为金钱。

下面我们逐条展开论述。

利用通才

在敏捷数据科学中，我们更看重通才而非专才，见图 1-9。

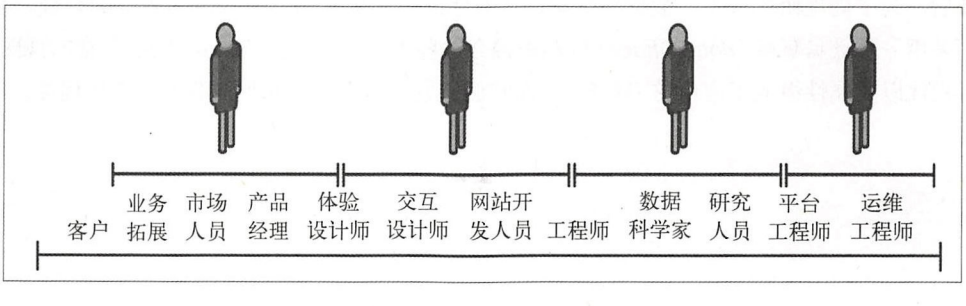


图 1-9　数据科学团队中宽泛的角色

换句话说，我们把团队成员技能的广度和他们知识储备的深度，以及其他领域的技能看得同样重要。敏捷数据科学团队中的优秀成员包括：

- 能写 CSS 的设计师。
- 能构建整套应用，并且理解用户接口和用户体验的网站开发人员。
- 既能做研究，又能开发网络服务和网络应用的数据科学家。
- 能贡献代码、解释结果、分享阶段性数据的研究人员。
- 掌握各处细节的产品经理。

敏捷数据科学团队中的设计尤其是一个关键角色。设计不仅限于外观和体验，而是包含产品的方方面面，从架构到发行，从用户体验到运行环境。

在纪录片《遗失的访谈》中，史蒂夫·乔布斯这样评论设计："设计产品就是在脑中记住五千个东西，以全新的不同以往的方式把它们组合在一起，以获得你想要的东西。每天你都会发现把这些东西稍稍变化后合到一起所产生的新问题或者新机会。这个过程就是魔法。"

使用敏捷的软件平台

在敏捷数据科学中，我们使用最简单易用、最方便的分布式系统，以及云计算和平台即服务（PaaS）等技术，最小化基础架构的成本，最大化生产效率。软件栈的简单性帮助我们重得敏捷性。使用这个软件栈，我们可以以尽可能少的步骤组建具有伸展性的系统。这让我们能行动迅速并吃进所有可用数据，而不会陷入伸展性问题而迫使我们丢掉部分数据或者火线重写程序。也就是说，程序只需一次编译，自动适应。

分享阶段性结果

最后，为了处理研究人员、数据科学家，还有团队中其他成员之间在时间表上的差异，我们采用一种数据拼贴（*data collage*）机制来融合这些不统一的节奏。换句话说，丰富的视图、可视化以及属性组成了应用的"菜单"，而我们把它们组合在一起形成我们的应用程序。

研究人员和数据科学家的工作周期比敏捷冲刺一般允许的最长周期还要长。他们每天都会产生很多数据，尽管这些数据的状态并不适合发布。但是在敏捷数据科学中，没有不能发布的状态。团队中的其他成员即使不方便每天看到数据状态的更新，也务必每周都要看到。这种与研究人员的沟通对于团结团队、支持产品管理至关重要。

这意味着要发布阶段性的结果，也就是发布未完成的数据，并且分析结果的碎片。这些“线索”保持了团队的整体性。随着这些阶段性结果可以交互，每个人都能接触到数据的本质，看到研究进度，了解到如何把这些线索整合为有价值的功能。开发和设计都必须从这些现实情况出发。这种持续发布一开始可以在小范围内进行，随着可看性的提升再逐渐扩大受众群体（见图 1-10），但是引入客户必须要尽早。

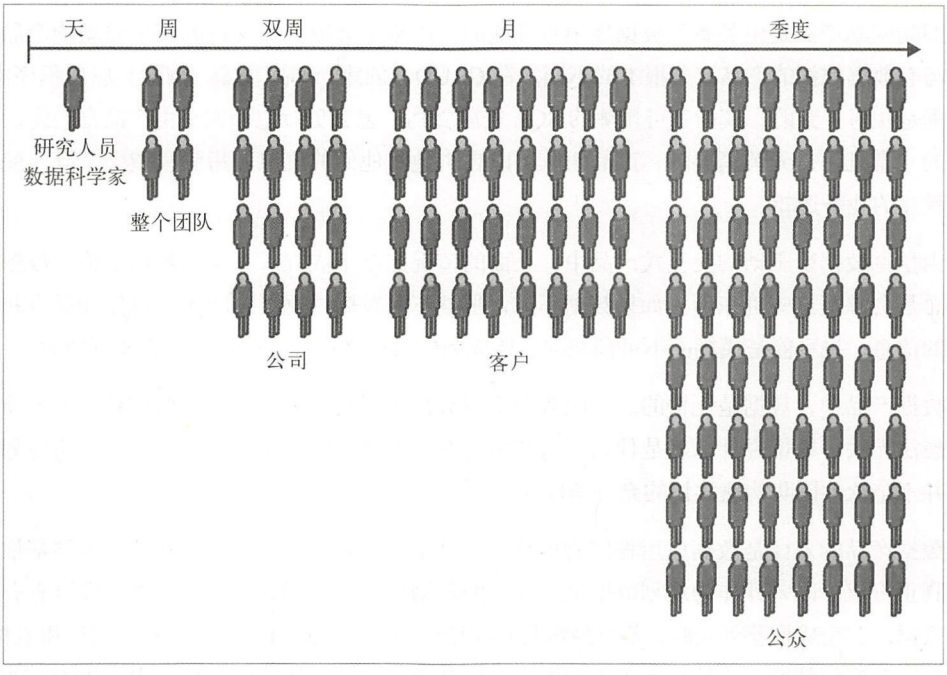


图 1-10　从概念到发布的过程中不断增长的受众

过程中的注意事项

敏捷数据科学过程抓住了数据科学中迭代的天性，以及所选工具赋予我们的高效率，从数据中构建并提取越来越高层的结构和价值。

考虑到数据科学团队内部广泛的技能，团队具有无穷的想象空间。团队跨越这么多学科，

使得建网站本质上是需要相互合作的。为了合作，团队需要方向，每个团队成员都要热情而又顽强地为之奋斗。为了朝着这个方向前进，团队需要达成共识。

在协作的同时建立和保持共识，是构建软件中最困难的部分。软件产品团队的主要风险是基于不同的规划各自为政。冲突的愿景导致意见不一致，最终会使产品失败。

有时，应用程序在做出来之前会被嘲笑：产品经理做了市场调研，而设计者却不断收到来自目标用户的差评。消解这些差评就是团队共同的规划。

即使数据是不变的，随着我们对用户增进了解以及客观环境的变化，真实的需求会不断发生变化。因此我们的规划也需要与时俱进。敏捷开发的初衷正是为了促进实现不断变化的需求，用真正有用的系统让差评消失。

典型的网站产品是由关系型数据库中可预测的、约束事务数据所支持的表单驱动的产品，这与有数据挖掘的产品有着根本的不同。在 CRUD（创建、读取、更新、删除）应用程序中，数据是相对一致的。模型是可预测的 SQL 表或文档，怎么更改它们完全由产品自己定。数据的“意见”是无关紧要的，产品团队可以自由地将他们的意愿应用到模型中，以匹配应用程序的业务逻辑。

在由挖掘数据所驱动的交互式产品中，上面的情况完全不成立。真实的数据很脏。数据挖掘总是会涉及这些脏东西。如果数据不脏，那就不叫数据挖掘了。即使经过仔细提取和精炼的信息，也可能是模糊而不可预测的。要成为能给最终用户看的信息，任重而道远。

在数据产品中，数据是无情的。无论我们希望数据告诉我们什么，结论都和我们先入为主的想法无关。数据说什么就是什么。这意味着瀑布模型无法使用，并且以差评作为规划依据并不是软件团队构建共识的充分条件。

对数据产品的差评是数据应用程序的规范，不过不包括数据产品的核心特性，也就是信息的真正价值。以差评作为规划依据是对复杂的数据模型做出了假设，但这些假设没有合理的依据。在生成推荐列表时，差评经常具有误导作用。鄙视完整的互动所起到的作用更糟：它们压制实际结论，支持先入为主的假设。但是我们知道，优秀的设计和用户体验应该是让先入为主的假设对实践的干扰最少。那我们怎么办？

敏捷产品开发的目标是认清应用的核心特性，优先构建这些部分，然后再加上其他功能。这为项目带来了敏捷性，使得项目更能满足真实而关键的需求，毕竟需求一直在演进。在数据产品中，这个核心特性会让你大吃一惊。否则，要么是你做得不对，要么是数据没有有意思的内涵。信息是有上下文的，当上下文可以互动时，能得出的见解也就不可预测了。

代码审核与结对编程

为了避免系统性的错误，数据科学家必须常规性地把自己的代码拿出来给其他团队的人看。这就体现出了正式代码审核的重要性。

要发现并修复代码的语法错误实在容易，独立发现算法中的系统性错误则要难得多，需要借助第二个、第三个乃至第四个人的审核。这些人也不需要都是数据科学家——只要数据科学家把代码的逻辑讲给别人听，任何一个程序员都可以发现代码实现中的偏差，并提出有帮助的建议。同时，规范正式的代码审核流程还有助于统一代码规范，提高代码可读性，有利于代码分享与解释。

如果没有代码审核，数据科学家可能会一直把时间浪费在改进一个南辕北辙的预测模型上。代码中系统性的错误极难被自己看出来，因为在阅读自己写的代码时，我们自然而然地跟着设想的逻辑走，而忽略了实际代码的逻辑。

每一轮冲刺中的代码审核都对保持高的代码质量和代码可读性至关重要。在算法层面上，避免系统性错误是非常重要的，这就需要团队形成一种总结和分享的风气。这种文化冲击是代码审核中最重要的一方面，因为这为团队成员提供了互相学习的机会，可以帮助团队成员熟悉理解那些他们平常不负责开发或维护的模块，并能修复其中的错误。如果一个关键的数据科学家或者数据工程师休病假了，你需要有人能发现并解决生产环境中的问题，这时你就知道建立代码审核流程是多么有先见之明。

敏捷开发的环境：提高生产效率

> 一排排的隔间如蜂巢密集。订满的会议室中人流川流不息。微软 Outlook 就是当代打卡机。彻底的疯狂。方块的海洋。
>
> 最后期限不断被大声喧嚣打断，空荡荡的桌子阻隔不了声波流转。无法工作，尽管人在眼前。噪声让人感觉远在天边。计划一直在变。
>
> 想提高生产效率的心如怪兽夺食，却无法落实。
>
> ——本书作者写的诗

通才比专才更需要不受打扰的安静环境，以便集中注意力。因为他们的工作内容涵盖更广，因此更需要全心投入。他们的工作环境必须满足这样的需求。

要么比传统的隔间工作区多投入两三倍的空间，要么就是让员工白费力气。敏捷开发的环境中，有些人不需要使用桌子，这能降低一些预算。

我们可以做得更好。我们应当做得更好。这会花更多的钱，但是花得值。

在敏捷数据科学中，我们不把团队成员当作普通的办公室白领，而是认可他们的创新能力。因此我们要构建的工作环境看起来更像是一个工作室，而非传统办公室。同时，我们认识到，使用高等数学探索数据需要安静的沉思和高度集中的注意力，因此我们也需要加入图书馆的元素。

> 许多企业把技能培训当作提高员工生产效率的唯一途径。然而，约 86% 的生产效率问题是与工作环境相关的。工作环境对员工的表现有很大影响。员工的工作环境决定了企业的天花板。
>
> ——Akinyele Samuel Taiwo

> 与运营一栋楼比起来，雇佣员工的开销要高得多，因此花钱提高工作环境是提高生产效率最经济的途径，毕竟少许的提高生产效率就能极大提高公司的利润。
>
> ——Derek Clements-Croome 和 Li Baizhan

创意工作者们需要三种不同的空间来相互合作，一起构建产品。按照从开发到封闭的顺序，这三种空间分别是：合作空间、个人空间、私有空间。

合作空间

合作空间是孵化点子的地方。合作空间应该在办公室内主要通道的沿线，在不同部门之间，宽敞明亮、开放、舒适而且迷人。这种空间不设墙，很灵活，可以随时调整。它们还在不断变化，时常重新布置，有很多懒人沙发、抱枕、舒服的椅子。合作空间要让人感觉到公司的正能量：笑声、高谈阔论，一个又一个议题让人感觉热血沸腾。把钱花在这里，并把这种空间开放展示。植物（除了塑料的假植物）可以用来隔音，还能制造新鲜空气！

私有空间

私有空间是在截止日期前赶工的地方。私有空间就像图书馆一样，封闭且隔音，不可以讲话。私有空间尽可能排除了一切引起分心的事物，想象一下淡淡的灯光和白噪音的感觉。私有空间里配有懒人沙发、躺椅以及椅子，也有完全符合人体工学设计的工作台。这类空间应该由帘子（珠帘）隔开，并且包含配有扩展坞的分别用来坐着办公和站着办公的书桌、30 英寸的专用显示器。

个人空间

个人空间也就像是自己的家。合作空间的开放度是最高的，而私有空间的私密性是最好的，

个人空间则是介于两者之间。个人空间是根据每个人的喜好分别布置的（比如选择共用办公室还是开放式办公桌、全包隔间还是半包隔间）。应该给大家提供菜单和预算，让大家自由发挥。我们也要鼓励空间主题设计以及布置绿色植物。这是大家在工作中待的时间最长的地方。另一方面，如果已经有了充足的合作空间和私有空间，个人空间里只需要一台笔记本电脑还有一部手机就足够了，有些人甚至压根都不需要这样的个人空间。

最重要的是，敏捷开发有这样的环境要求是为了营造出一种浸没在数据中的感觉，因为实际环境到处都是打印物、海报、书籍、白板等，见图 1-11。

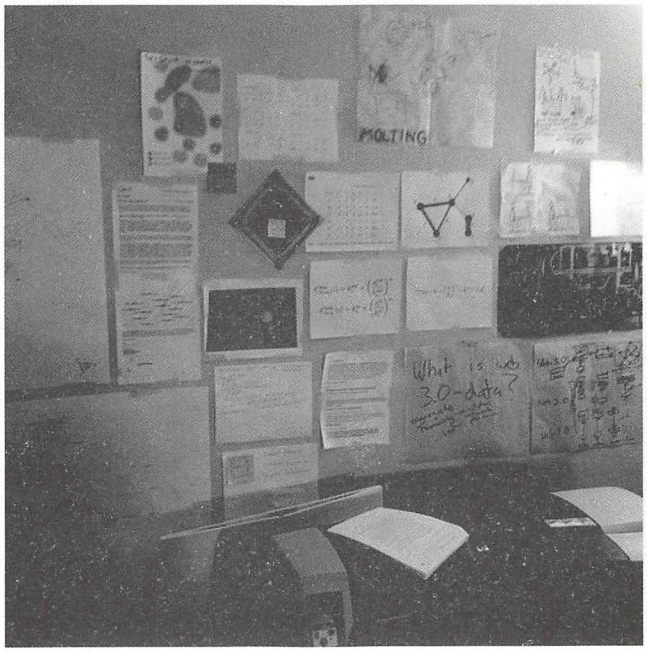


图 1–11　到处张贴让人感觉沉浸在数据中

如果团队获得了这三种环境，就会变得愉悦而高产，能够更高效地应对数据科学挑战。

用大幅打印实现想法

敏捷开发环境的要求包括能方便地进行大幅打印。可视化的物质形式能够促进思路分享、思想碰撞、想法表达以及创造性。

仅供非商业用途或交流学习使用

有几家公司生产的 24 英寸宽幅打印机售价低于 1000 美元。一套不间断供墨系统的价格则低于 100 美元。因此，以 24 英寸 ×36 英寸的海报打印为例，每打印一张海报的成本不到 1 美元。

考虑到价格成本并不高，我们没有理由不给数据科学团队随时进行大幅打印的权利，他们可以把平常的证明过程打印出来看，也可以打印一些色彩丰富的图文。当数据科学团队取得新进展时，很容易让人们对跨部门的数据感到兴奋。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

第 2 章

敏捷工具

本章简要介绍我们要用的软件栈，这些软件是专为我们的处理优选出来的。

等到读完本章，你已经可以对数据进行收集、存储、处理、发布和展示（见图 2-1）。这套软件栈可以让一个人完成全部这些工作，掌握“全栈”技术。

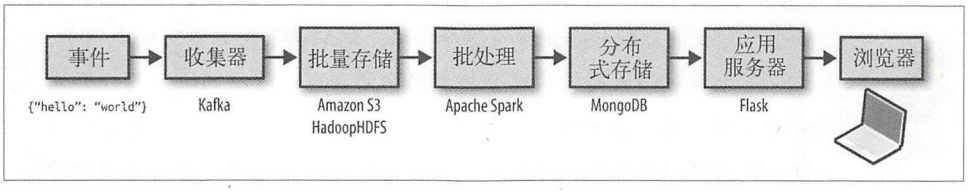


图 2-1　用软件栈处理数据

数据科学家离不开全栈技术。这里会很快地讲到其中许多技术，但是不用担心——在第 5 ~ 9 章中，我还会继续展示这套软件栈。在本章中你只需要理解一些基本知识，读了后面的内容后自然会理解很多。

我们先讲讲如何在你自己的电脑上的本地模式下运行整套软件栈。在第 3 章中，你将学到如何通过使用亚马逊云服务，使这套软件栈在云上运行。让我们开始吧！

本章的代码示例在 Agile_Data_Code_2/ch02（*https://github.com/rjurney/Agile_Data_ Code_2/tree/master/ch02*）中可以找到。复制代码仓库，跟着一起做吧！

```
    git clone https://github.com/rjurney/Agile_Data_Code_2.git
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

可伸缩性＝易用性

数据科学以及大数据的发展中，人们在钻研分析应用上投入了许多精力，开发出了许多NoSQL 工具，比如 Spark、Hadoop、MongoDB 等。然而，本书并不是一本介绍大数据基础架构的书。本书的目的是要教你如何使用这些基础架构软件来构建出自己的应用。等我们介绍完了这套软件栈，就可以只用关心如何使用和依赖这套系统构建我们自己的应用了。因此，本书只会用两章的篇幅来介绍基础架构——一章用来介绍我们的开发工具，另一章则介绍如何在云上使用这些工具，以满足数据规模方面的要求。

在选择工具的时候，我们希望能获得线性、水平的伸缩性，但是，最重要的还是易用性。让现代分析应用适用于各种数据规模所需的并发系统实现起来很复杂，我们还是需要能专注于手头的任务：处理数据并为用户创造价值。当工具过于复杂，很多配置项没有足够好的默认值而需要人工调整时，我们就过于关注工具本身了。我们应该关注的是数据、用户，以及有用的新应用。为了实现这样的目的，我们要的是一套简单的软件栈。这样一个高效的软件栈可以让包括设计、应用开发、统计、机器学习在内的各团队轻松合作，而无须分布式系统的专家参与其中。

本书中列出的软件栈不是唯一选择。这是端到端解决方案的一个例子，足以用来应对快速高效构建分析型应用的要求。这样的方案符合开发者的期望，也应当是基础架构工程师的目标。读者应该会获得一个可以快速启动开发的软件栈实例，以及自己搭配软件栈时需要遵循的准则。

敏捷数据科学之数据处理

构建分析型应用的第一步是把应用从头到尾的数据流动通道打通，包括从原始数据一直到用户屏幕上的显示内容（见图 2-2）。这很重要，因为复杂度可能会变得很高，你需要从一开始就考虑用户反馈，而不是不顾反馈地迭代（也被称为死亡循环（*death spiral*））。

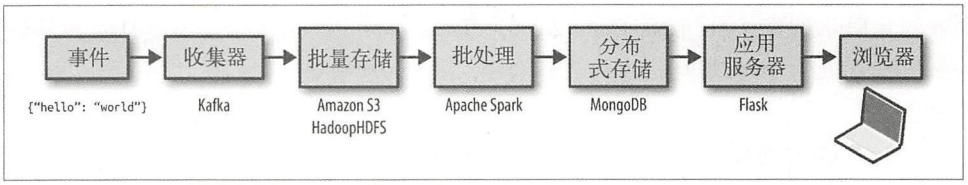


图 2-2　本书软件栈中的数据处理流程图

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

我们软件栈的组成部分如下：

- 事件（*event*）就是日志所代表的东西。我们把发生的记录了特征和时间戳的一次经历称为一个事件。

  事件以多种形式出现，比如服务器端的日志、传感器、财务方面的一笔交易，还有用户在应用程序中的行为。为了便于在各种工具和编程语言之间进行数据交换，我们要把事件序列化为约定的通用格式。

  在本书中，我们使用 JSON 行文件（JSON Line）（*http://jsonlines.org/*）序列化数据，这是一种把每个 JSON 对象存储一行文本，中间用回车隔开的简单格式。JSON 行文件使用扩展名 *.jsonl*。而我们经常用 gzip 进行压缩，这样文件名后缀就成了 *.jsonl.gz*。

  当我们需要考虑性能的时候，我们使用列式存储格式 Apache Parquet（*https://parquet.apache.org/*）。Parquet 是适用于多种语言和工具的跨平台文件格式。从 Parquet 文件中读取几行记录要比从压缩的 JSON 文件中读取快得多。

- 收集器（*collector*）是用来聚合事件的。它从一个或者多个数据源中收集事件，聚合后记录到批量存储中或者写入实时处理队列等待操作。Kafka 是现在主流的使用批量存储的事件聚合解决方案。
- 批量存储（*bulk storage*）是一种能为许多并发进程并行访问的场景提供高 I/O 支持的文件系统（想象有很多磁盘或者 SSD）。我们将使用 S3 来替代 Hadoop 分布式文件系统（HDFS）。HDFS 是批量存储的标杆，没有 HDFS 就没有大数据。对于我们在敏捷数据科学中处理数据所需的高 I/O 吞吐量和庞大的数据量，用 HDFS 太贵了。
- 分布式文档存储（*distributed document store*）是使用文档格式的多节点存储。在敏捷数据科学中，我们用它来发布数据给网络应用和其他服务使用。我们将使用 MongoDB 作为我们的分布式文档存储系统。许多人不考虑 MongoDB，因为他们用了 MongoDB 的很多功能之后发现它和其他数据库一样有伸缩性方面的问题。不过，仅仅作为文档存储使用时（读取文档，而不做聚合或者其他种类的查询），MongoDB 的伸缩性不输于其他系统。我们避免使用其他那些功能就行了。
- 轻量级的网站应用服务器（*application server*）让我们以最小的开销，通过可视化客户端获取 JSON 格式的数据。我们使用 Python/Flask，这样读者不用多学一门编程语言。我们还可以用 Python 的 *sklearn* 和 *xgboost* 来部署我们的机器学习服务。轻量级的网站开发框架还有 Ruby/Sinatra 和 Node.js 等。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

- 现在基于浏览器（*browser*）的应用和手机应用可以让我们把数据以交互式的形式展现给用户，而用户又通过交互和交互所产生的事件为我们提供了数据。在本书中，我们只讨论网络应用程序。

这个列表看起来可能比较长，令人望而生畏，但是实际上这些工具都很容易安装，而且和数据科学中的关键点相匹配。图 2-3 展示了整体架构。这样的配置可以轻松伸缩，并针对分析处理进行了优化。

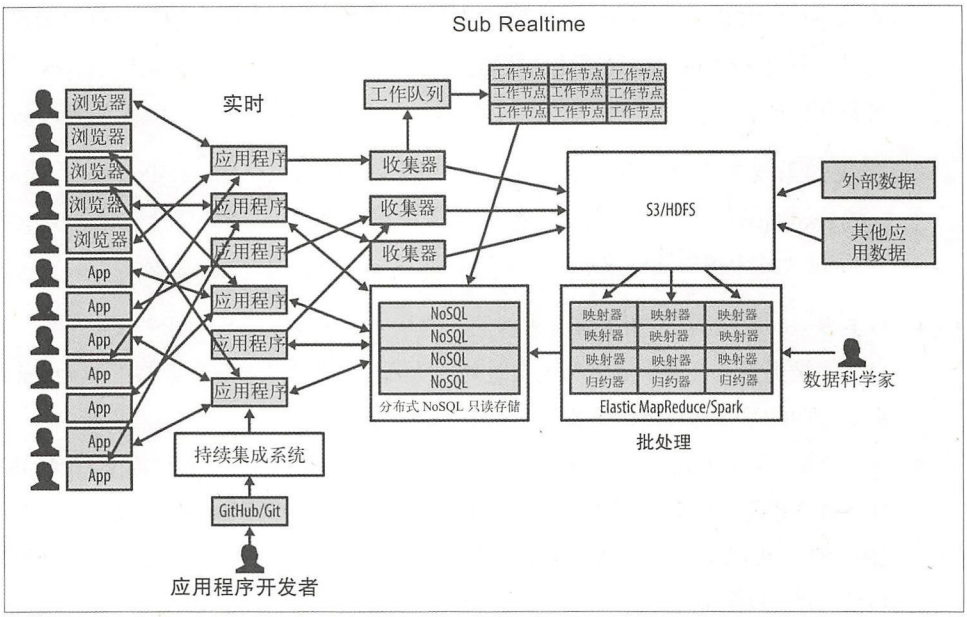


图 2–3　整体架构

搭建本地环境

要安装软件来搭建本书要用的环境有几种方式。你可以装在虚拟机上，也可以装在自己的电脑上，还可以使用 Amazon Web Services（AWS）。**推荐使用 EC2 环境来运行书中的示例代码。**

在本节中，我们会介绍如何在电脑上搭建虚拟机（VM）来运行示例。如果你想直接装在电脑上，可以阅读附录 A 后手动安装这些工具。我推荐使用 Vagrant 或者 AWS，因为这样会简单得多，不过我是在自己的 MacBook Pro 上直接使用这些工具的。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

配置要求

你需要为 Vagrant/VirtualBox 虚拟机保留 9 GB 以上的空闲内存来运行最吃内存的几个示例（第 7、8、9 章中的模型拟合）。建议先把一些不需要的程序关掉，然后重启电脑，再启动 Vagrant 虚拟机。如果你的机器没法满足这样的要求，我建议使用 Amazon Web Services，参见第 2 章中“搭建 EC2 环境”一节的内容。

配置 Vagrant

Vagrant（*https://www.vagrantup.com/*）可以让我们创建并配置轻量级、可重现的便携开发环境。本书写作时，Vagrant 的最新版本是 1.9.3。你可以在它的下载页面（*https://www.vagrantup.com/downloads.html*）上找到安装说明的链接。

你需要有 VirtualBox（*https://www.virtualbox.org/*）才能使用 Vagrant。安装指导可以在 VirtualBox 用户手册（*https://www.virtualbox.org/manual/ch02.html*）中找到。

注意：如果你已经安装好了 VirtualBox，可能要更新到最新版本才能运行 Vagrant 环境。现在就更新吧。

本书代码仓库中的 Vagrantfile 文件已经配置好了，可以通过如下命令使用：

```
vagrant up
```

执行需要花一些时间。执行好之后，可以使用如下命令连接：

```
vagrant ssh
```

示例代码在 *Agile_Data_Code_2* 目录中。你需要把工作目录切换（cd）到这个文件夹中来运行示例代码。如果 Vagrant 的用户目录中没有 *hadoop*、*spark*、*kafka* 以及 *Agile_Data_Code_2* 这些文件夹，请等待几分钟让引导脚本完成处理。

下载数据

需要运行脚本 *download.sh*（*https://github.com/rjurney/Agile_Data_Code_2/blob/master/download.sh*）来下载本书使用的示例数据集。脚本会把数据存储在 *Agile_Data_Code_2/data/* 子目录中。如果要直接跳到第 8 章，则需要运行 *ch08/download_data.sh*（*https://github.com/rjurney/Agile_Data_Code_2/blob/master/ch08/download_data.sh*）。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

搭建 EC2 环境

脚本 *ec2.sh*（*https://github.com/rjurney/Agile_Data_Code_2/blob/master/ec2.sh*）可以用来启动 EC2 实例，配置好项目环境并安装代码。你需要使用 Amazon Web Services Command Line Interface（*https://aws.amazon.com/cn/cli/*）（AWS CLI）来运行脚本，它可以通过 Python 的 `pip` 命令安装：

```
pip install awscli
```

安装好 AWS CLI 之后，请查看 *ec2.sh*（*https://github.com/rjurney/Agile_Data_Code_2/blob/master/ec2.sh*）文件。它会启动一个 `r3.xlarge` 类型的实例，使用 *aws/ec2_bootstrap.sh*（*https://github.com/rjurney/Agile_Data_Code_2/blob/master/aws/ec2_bootstrap.sh*）安装软件环境并检查示例代码。在本书写作时，这种实例的价格是每小时 0.266 美元，所以你可能需要在实践的间隙关掉实例。

使用 *ec2.sh* 要用到 `jq` 工具（*https://stedolan.github.io/jq/*），它可以解析 `aws` 命令返回的 JSON 响应消息。*ec2.sh* 会尝试借助脚本 *jq_install.sh*（*https://github.com/rjurney/Agile_Data_Code_2/blob/master/jq_install.sh*）所用平台的软件包管理器来安装 `jq`。如果没能自动安装，脚本会给出 `jq` 的安装页面（*https://github.com/stedolan/jq/wiki/Installation*），以便自行安装。等 `PATH` 中有了 `jq` 命令，再次执行 *ec2.sh* 脚本，就可以继续执行了。

ec2.sh 创建出了名为 *agile_data_science* 的密钥对，保存在 *agile_data_science.pem* 中。然后它创建了一个名为 `agile_data_science` 的安全组，设置仅允许你的外部 IP 访问 22 端口 SSH 服务。这样，除你自己电脑以外的电脑就都连不上该实例了。脚本会使用创建出来的密钥对和安全组来启动 `r3.xlarge` 实例。

你可以在 Amazon EC2 控制台（见图 2-4）上看到脚本启动的机器。确保 URL 中的地区（例如 *us-west-2*）和你通过 `aws` 命令配置的默认地区一样，不然看不到任何主机实例。主机名字叫 `agile_data_science_ec2`。如果不确定 `aws` 命令所配置的地区是哪个地区，请输入 `aws configure` 并在输出中查看地区信息。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

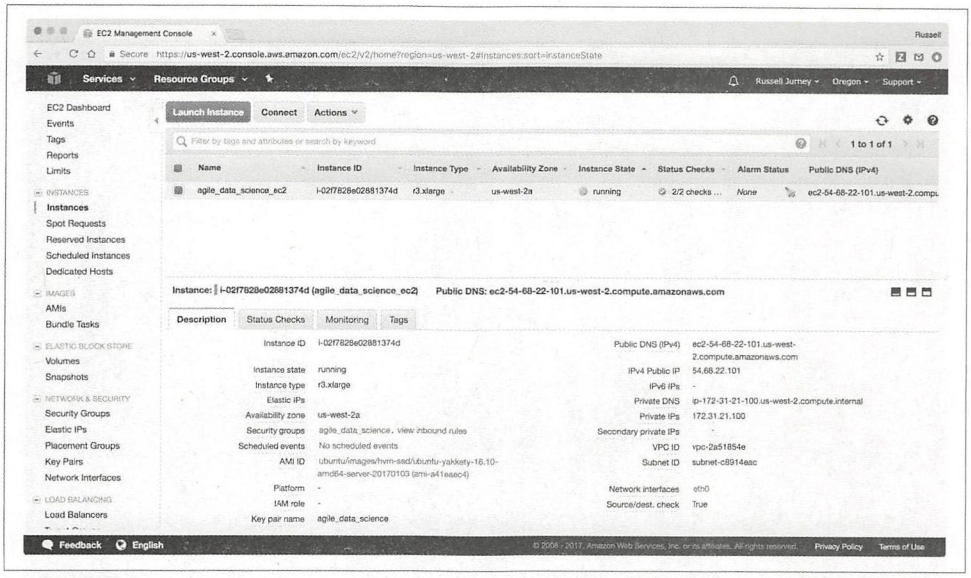


图 2-4　EC2 控制台关于所启动实例的详细描述

执行完成后，脚本会用红色字体标明登录用的 SSH 命令，同时告诉你登录还要再等几分钟，因为机器正在初始化（见图 2-5）。几分钟过后，运行脚本 *ec2_create_tunnel.sh*（*https://github.com/rjurney/Agile_Data_Code_2/blob/master/ec2_create_tunnel.sh*）来创建 SSH 隧道，将本地端口 5000、8080、8888 监听到的请求转发到 EC2 实例上（见图 2-6）。这样你就可以在本地机器上通过 *http://localhost:5000* 访问 EC2 实例上运行监听 5000 端口的网络应用，通过 *http://localhost:8888* 访问 EC2 实例上运行的 Jupyter 笔记本，通过 *http://localhost:8080* 访问 EC2 上的 Apache Airflow 接口。

仅供非商业用途或交流学习使用

```
2. bash
Russells-MacBook-Pro-OLD-506:Agile_Data_Code_2 rjurney$ ./ec2.sh

Welcome to Agile Data Science 2.0 :)

I will launch an r3.plarge instance in the default VPC for you, using a key and security group we will create.

The utility 'jq' is required for this script to detect the hostname of your ec2 instance ...
Detecting 'jq' ...
'jq' was detected ...
Creating security group 'agile_data_science' ...
{
    "GroupId": "sg-f21b248a"
}

Detecting external IP address ...
Authorizing port 22 to your external IP (73.92.31.248) in security group 'agile_data_science' ...

Generating keypair called 'agile_data_science' ...
Changing permissions of 'agile_data_science.pem' to 0600 ...

Detecting the default region...
The default region is 'us-west-2'
Determining the image ID to use according to region...
The image for region 'us-west-2' is 'ami-a9lacac4' ...

Initializing EBS optimized r3.xlarge EC2 instance in region 'us-west-2' with security group 'agile_data_science', key name 'agile_data_science' and image id 'ami-a9lacac4' using the script 'aws/ec2_bootstrap.sh'
Got reservation ID 'r-0cb19874f8323a1c7' ...

Sleeping 10 seconds before inquiring to get the public hostname of the instance we just created ...
...
Awake!

Using the reservation ID to get the public hostname ...
The public hostname of the instance we just created is 'ec2-52-41-69-55.us-west-2.compute.amazonaws.com' ...
Writing hostname to '.ec2_hostname' ...

After a few minutes (for it to initialize), you may ssh to this machine via the command in red:

Note: only your IP of '73.92.31.248' is authorized to connect to this machine.
Note: if you ssh to this machine after a few minutes and there is no software in $HOME, please wait a few minutes for the install to finish.

Once you ssh in, the exercise code is in the Agile_Data_Code_2 directory! Run all files from this directory, with the exception of the web applications, which you will run from ex. ch08/web

Note: now you will need to run ./ec2_create_tunnel.sh to forward port 5000 on the ec2 instance to your local port 5000. This way you can run the example web applications on the ec2 instance and browse them at http://lo
calhost:5000

--------------------------------------------------------------------------------

Thanks for trying Agile Data Science 2.0!
If you have ANY problems, please file an issue on Github at https://github.com/rjurney/Agile_Data_Code_2/issues and I will resolve them.

If you need help creating your own applications, or with on-site or video training...
Check out Data Syndrome at http://datasyndrome.com

Enjoy! Russell Jurney <@rjurney> <russell.jurney@gmail.com> <http://linkedin.com/in/russelljurney>
Russells-MacBook-Pro-OLD-506:Agile_Data_Code_2 rjurney$
```

图 2–5　执行 *ec2.sh*

在用完了 EC2 实例后或是希望端口在下次使用前恢复原样，请通过 *ec2_kill_tunnel.sh*（*https://github.com/rjurney/Agile_Data_ Code_2/blob/master/ec2_kill_tunnel.sh*）脚本关闭 SSH 隧道。你可以随时通过运行 *ec2_create_tunnel.sh*（*https://github.com/rjurney/Agile_Data_Code_2/blob/master/ec2_create_tunnel.sh*）脚本再次打开 SSH 隧道进行端口转发。

```
2. ubuntu@ip-172-31-23-26: ~ (bash)
BOOK (bash)  ⌘1   ubuntu@ip-172-31... ⌘2

Russells-MacBook-Pro-OLD-506:Agile_Data_Code_2 rjurney$ ./ec2_create_tunnel.sh

This script will create an ssh tunnel between the ec2 host's port 5000 and your local port 5000.
This will enable you to view web applications you run from ex. Agile_Data_Code_2/ch08/web to be viewed at http://localhost:5000
Note: the tunnel will run in the background, and will die when you terminate the EC2 instance.

First we will forward the port the web applications use...
Forwarding the remote machine's port 5000 to the local port 5000, which you can then access at http://localhost:5000
Executing: ssh -N -i ./agile_data_science.pem -o StrictHostKeyChecking=no -L 5000:localhost:5000 ubuntu@$EC2_HOSTNAME &

Next we will forward the port the Jupyter Notebooks use...
Forwarding the remote machine's port 8888 to the local port 8888, which you can then access at http://localhost:8888
Executing: ssh -N -i ./agile_data_science.pem -o StrictHostKeyChecking=no -L 8888:localhost:8888 ubuntu@$EC2_HOSTNAME &

Russells-MacBook-Pro-OLD-506:Agile_Data_Code_2 rjurney$
```

图 2–6　执行 *ec2_create_tunnel.sh*

使用 SSH 登入主机后，屏幕上会出现下一步的操作说明（见图 2-7）。如果没看到这段文字，说明主机没配置好。请断开连接，等待几分钟后再进行连接，让启动脚本能执行完成；这样，你就能看到操作说明了。

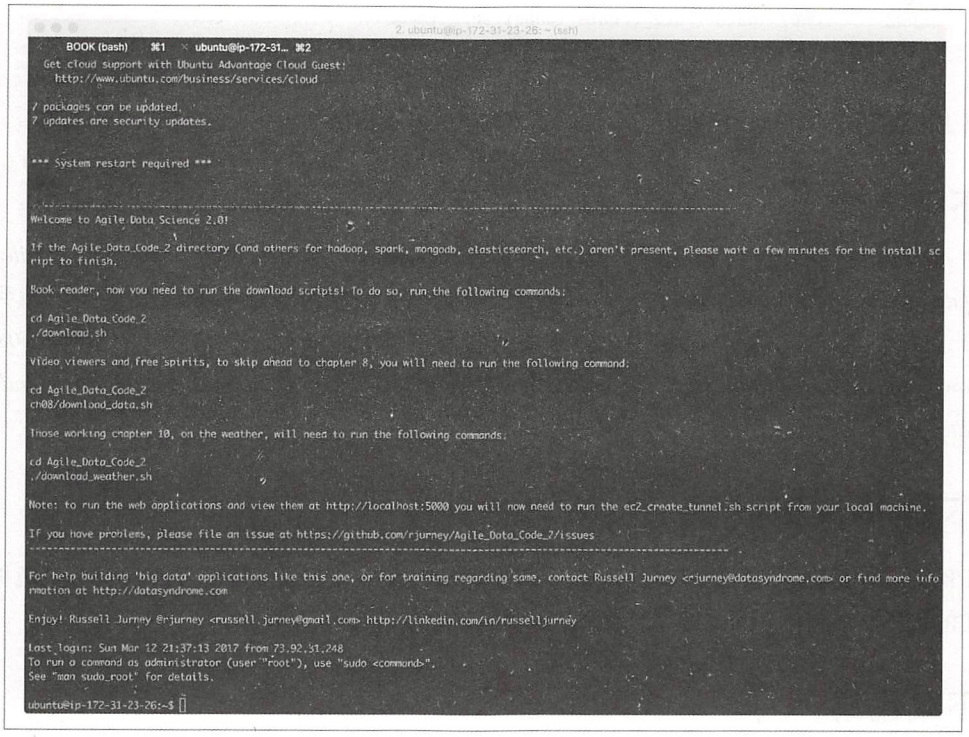


图 2-7　使用机器 *agile_data_science_ec2* 的操作说明

登录进去之后，先列出家目录的内容，你可以看到示例代码目录和构成我们开发环境的全部软件：

```
$ls
Agile_Data_Code_2  agile_data_science.message  airflow  anaconda  elasticsearch
 elasticsearch-hadoop  hadoop  kafka  logs
 spark  zeppelin
```

现在把路径切换到示例代码目录 *Agile_Data_Code_2* 中，列出其中的内容：

```
$ cd Agile_Data_Code_2
$ls
aws            ch05  ch09        download.sh              ec2.sh
jq_install.sh            manual_install.sh        spark-warehouse
bootstrap.sh  ch06  ch10        download_weather.sh     elastic_scripts
jupyter_notebook_config.py   models              Vagrantfile
ch02       ch07  data   ec2_create_tunnel.sh   images              lib
README.md   ch04         ch08  Dockerfile        ec2_kill_tunnel.sh
intro_download   .sh  LICENSE          requirements.txt
```

马上就可以开始运行示例程序了。但是还要先下载好数据！

下载数据

通过 SSH 连上实验机器之后，运行脚本 *download.sh*（*https://github.com/rjurney/Agile_Data_Code_2/blob/master/download.sh*）来下载本书示例要用的数据集。数据会下载到目录 *Agile_Data_Code_2/data/* 中。如果要直接跳到第 8 章开始，则需要运行脚本 *ch08/download_data.sh*（*https://github.com/rjurney/Agile_Data_Code_2/blob/master/ch08/download_data.sh*）。

下载并运行代码

示例代码都是现成的，你可以试着运行并修改这些代码，最终把它们改成使用你自己数据集的代码，放到自己的应用程序中去。

下载代码

除 EC2 或者 Vagrant 映像中的代码外，你还需要在本地放一份代码用来阅读、修改或者随便玩玩。你可以从 Github 上克隆代码仓库，通过如下命令查阅：

```
$ git clone https://github.com/rjurney/Agile_Data_Code_2
$ cd Agile_Data_Code_2
$ ls
```

运行代码

示例代码预期的工作路径是 *Agile_Data_Code_2* 一级目录，而不是每章的子目录。这条法则不适用于网络应用的代码，那些代码应当在章节目录的 web 子目录中运行（例如 *ch08/web*）。

Jupyter 笔记本

你需要从项目根目录 *Agile_Data_Code_2* 上运行 Jupyter 笔记本。如果你用的是配置好的 Vagrant 或者 EC2，那么启动脚本已经帮你启动了 Jupyter 笔记本，你可以直接通过 *http://localhost:8888* 连上去。我们稍后还会进一步讨论 Jupyter 笔记本。

工具集概览

如果你喜欢边做边学，则可以先略读本章余下的内容，然后直接从第 3 章开始阅读。在这一节中，我们会介绍本书中要用到的工具，用每个工具跑一个最简单的例子，然后介绍怎样用它们构建出一个完整的系统。如果你想了解这些工具的安装细节，请参阅附录 A。

敏捷开发工具栈的要求

对于数据科学技术栈来说，为了实现敏捷性，有那些必需的要求？

一个要求是栈的每一层都要水平可伸缩。往集群中再加一台机器比升级昂贵的专有硬件要好得多。如果要重写预测模型的实现才能重新部署，这就不敏捷了。这就是为什么我们要使用 Spark MLlib 而不是那些专门为单机设计的工具。

另一个要求是在栈的各层之间上下传递数据必须要能一行代码解决。在今天的配置密集型环境中，这是一个比较高的要求，但是我们可以通过精心挑选工具来满足。总的来说，这些要求可以让我们在大规模集群上保持高产出。

Python 3

在本书写作中，我使用了 Python 3，而且我强烈推荐你们也用它。我们提供的 Vagrant 与 EC2 的映像都已经预装了 Python 3，所以如果你用映像的话，就不用安装了。

你也可以使用 Python 2.7，但有时候示例代码可能会无法运行，这时需要按 2.7 的语法修改异常处理的格式。这是我们在大部分地方唯一使用的 Python 3 特有的语法。还有就是在第 8 章中，我们在 Kafka API 中使用了 bytes 类型而非字符串类型，这也是 Python 3 所特有的。使用 Python 2.7 也许还会遇到一两个错误，不过应该都可以很容易地通过极少的改动解决。话虽如此，我依然强烈推荐使用 Python 3。

由于 Spark 2.1.0 无法与 Python 3.6 兼容，所以我们在本书中使用了 Python 3.5。本书原版出版后不久 Spark 2.1.1 就发布了，后续版本已经兼容于 Python 3.6。

Anaconda 与 Miniconda

在本书中，我们使用 Anaconda Python 3.5，因为 Anaconda 已成为数据科学领域领先的 Python 发行版。Anaconda 是 Continuum Analytics 出品的 Python 发行版，包含超过 400 个流行的数据科学库。编译安装 `numpy` 和 `scipy` 等库是比较麻烦的，Anaconda 提供了一条捷径。

尽管我推荐在你们自己的电脑上使用功能齐全的 Anaconda，在 Vagrant 和 EC2 映像中我使用的是 Anaconda 的小弟弟 Miniconda。这是因为 Anaconda 太大了，下载会花很长时间（20 ~ 30 分钟）。而下载 Miniconda 只要几分钟。Miniconda 和 Anaconda 很像，只是少装了一些包。幸运的是，`conda` 和 `pip` 工具可以让我们迅速安装我们需要的包，给了我们一个符合需求而简化的 Python 3 发行版。

Jupyter 笔记本

在第 7 章和第 9 章中我们会利用 IPython/Jupyter 笔记本交互式地使用 Python，进行数据可视化、训练预测模型并改进。Jupyter 笔记本让我们可以在网页上分享分析过程，用保存的变量、图表和数据表格来完成。

为什么不试试 Jupyter 笔记本，熟悉一下用法呢？如果你用的是 Vagrant 或 EC2 映像，那么 Jupyter 笔记本已经在项目根目录中运行了，你可以直接访问 *http://localhost:8888*。

这会打开一个列出了示例代码目录 *Agile_Data_Code_2* 中文件的窗口。选择 New → Python 3（见图 2-8）。

仅供非商业用途或交流学习使用

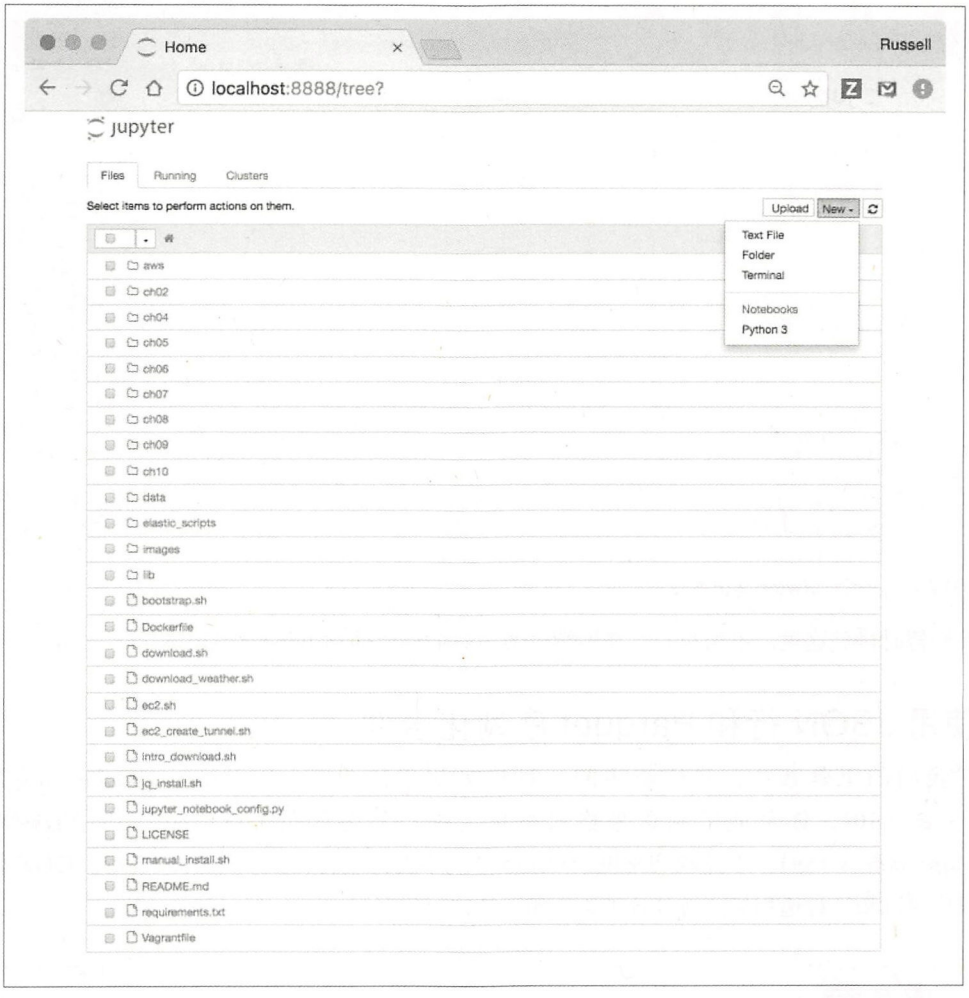


图 2-8　Jupyter 主页面

这会在浏览器的新标签页中打开一个 Jupyter 笔记本（见图 2-9）。输入 `print("Hello, World!")`，然后单击“执行”按钮。网页上的 Python——很酷，对吧？

仅供非商业用途或交流学习使用

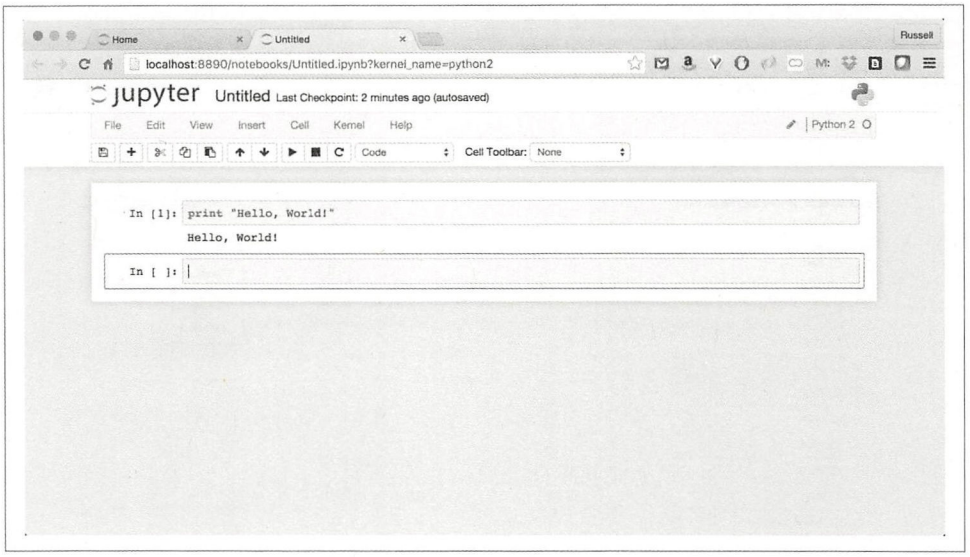


图 2-9　一个 Jupyter 笔记本

我们暂时讲到这里。不用担心，在后续章节中我们还会提到 Jupyter 笔记本。

使用 JSON 行和 Parquet 序列化事件

在我们的工具栈中，我们使用的一种序列化系统为 JSON 行（*http://jsonlines.org/*）（见图 2-10）。你可能听到这种格式被称为换行符分隔的 JSON 格式，即 NDJSON（*http://ndjson.org/*），不过确切来说 JSON 行格式不支持空行，而 NDJSON 支持。JSON 使我们可以用一种通用格式通过各种语言和工具访问数据。

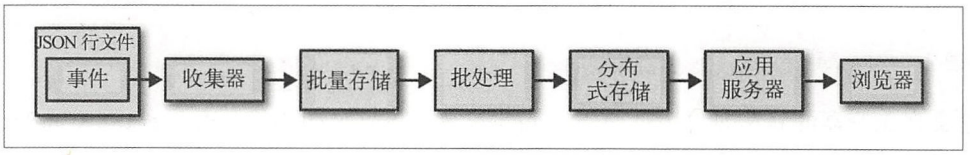


图 2-10　序列化事件

但是 JSON 行并不适用于所有的用例——尤其是性能比较重要而数据是表格状时。在这些情况下，我们使用 Apache Parquet（*https://parquet.apache.org/*）格式。Parquet 是一种跨平台数据格式，使用 Parquet 格式读取数据中的几列要比读取整行数据高效得多。这可以让我们的分析实时完成。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

放弃 Avro

本书第 1 版使用了 Avro 作为序列化方式，但是我已经把所有的数据处理改为使用回车分隔的 JSON（JSON 行 /NDJSON），因为在使用 Avro 时我经常遇到 Avro 的某个库中的错误，令我后悔不已，而使用 JSON 格式时从未这样。大多数编程语言都原生支持 JSON。它是所有格式中最适合用来构建分析型应用的。

从技术上来说，对于许多类型的数据，Avro 能比 JSON 更高效地进行序列化。比如，在编码图片、非 UTF-8 文本、二进制块时，Avro 就更高效。然而，键 / 值对或块存储才是图片和数据块应该待的地方，在 JSON 文件中引用它们才是最好的办法。非 UTF-8 字符串需要在序列化前转为 UTF-8，而 Avro 中的 Unicode 编码非常难用（*https://issues.apache.org/jira/browse/AVRO-565*）。Avro 比 JSON 功能多，不过这恰恰是我所遇到的问题——Avro 做了一些更应该在别处处理的事情。由于我们可以选择要用的架构（很多人不这么做），故 JSON 从一开始就是最好的选择。这不是说 Avro 是不好的文件格式，它其实很好！考虑到我们重视简单程度，而 JSON 远比 Avro 简单。对不起，Doug[1]，我们依然爱你！

Python 中的 JSON

`json` 模块在 Python 2.7 和 3.x 的标准库中（*http://bit.ly/1upkGOV*）不需要额外安装。要读取和写入 JSON 行文件，我们只需要短短几行代码。参见 ch02/test_json.py（*http://bit.ly/2oCtUxR*）：

```
#
# 如何使用 Python 读写 JSON 和 JSON 行文件
#
import sys, os, re
import json
import codecs

ary_of_objects = [
  {'name': 'Russell Jurney', 'title': 'CEO'},
  {'name': 'Muhammad Imran', 'title': 'VP of Marketing'},
  {'name': 'Fe Mata', 'title': 'Chief Marketing Officer'},
]

path = "/tmp/test.jsonl"
#
```

1 Doug 指 Doug Cutting，他是 Avro 的发起人，也是“Hadoop 之父”。——译者注

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

```
# 把对象写为 jsonl 文件
#
f = codecs.open(path, 'w', 'utf-8')
for row_object in ary_of_objects:
  # ensure_ascii=False is essential or errors/corruption will occur
  json_record = json.dumps(row_object, ensure_ascii=False)
  f.write(json_record + "\n")
f.close()

print("Wrote JSON Lines file /tmp/test.jsonl")

#
# 把 jsonl 文件读取回对象
#
ary_of_objects = []
f = codecs.open(path, "r", "utf-8")
for line in f:
  record = json.loads(line.rstrip("\n|\r"))
  ary_of_objects.append(record)
print(ary_of_objects)
print("Read JSON Lines file /tmp/test.jsonl")
```

我写了几个帮助函数，隐藏了这些操作的具体细节：

```
import codecs, json

def write_json_file(obj, path):
  ''' 转存一个对象并以 JSON 格式写入文件 '''
  f = codecs.open(path, 'w', 'utf-8')
  f.write(json.dumps(obj, ensure_ascii=False))
  f.close()

def write_json_lines_file(ary_of_objects, path):
  ''' 转存对象列写入 JSON 行文件 '''
  f = codecs.open(path, 'w', 'utf-8')
  for row_object in ary_of_objects:
    json_record = json.dumps(row_object, ensure_ascii=False)
    f.write(json_record + "\n")
  f.close()

def read_json_file(path):
  ''' 把普通 JSON 文件（没有换行符分隔记录）转为对象 '''
  text = codecs.open(path, 'r', 'utf-8').read()
  return json.loads(text)

def read_json_lines_file(path):
  ''' 把 JSON 行文件（每条记录间用换行符隔开）转为对象数组 '''
  ary=[]
  f = codecs.open(path, "r", "utf-8")
  for line in f:
    record = json.loads(line.rstrip("\n|\r"))
    ary.append(record)
  return ary
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

验证记录文件存在：

```
$ ls -lah /tmp/test.jsonl
-rw-r--r-- 1 rjurney wheel 154B Mar 17 17:19 /tmp/test.jsonl
```

检查我们写入的文件内容，目测是正确的：

```
$ cat /tmp/test.jsonl
{"name": "Russell Jurney", "title": "CEO"}
{"name": "Muhammad Imran", "title": "VP of Marketing"}
{"name": "Fe Mata", "title": "Chief Marketing Officer"}
```

看起来一些都很正常！我们后面还会用到这些帮助函数，你可以在 utils.py（*https://github.com/rjurney/Agile_Data_Code_2/blob/master/lib/util.py*）中找到它们以及其他一些全书都会用到的工具函数。就是这样！在 Python 中使用 JSON 行几乎不费吹灰之力。

收集数据

除了用于实时作业，Kafka（见图 2-11）也成为了进行数据混洗的一种不错的选择。对于敏捷数据科学要做的事情来说，我们需要访问日志以及分布式文件系统上的其他数据。然而就开发而言，还是本地工作更简单。所以，本书的示例将使用本地文件系统，我们的数据收集工作基本上只包括下载文件然后进行本地处理。在生产环境中，我们可能要从 Kafka 或者 Amazon Kinesis 中收集事件，并同步到 S3 以进行批处理。

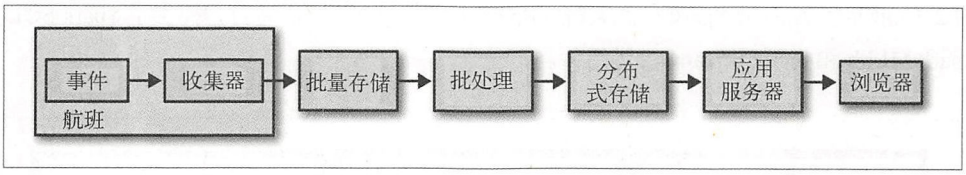


图 2-11　使用 Kafka 收集数据

使用 Spark 进行数据处理

Spark 是领先的分布式通用数据处理平台。Spark 把数据处理切分为小任务交给一群廉价的 PC，每个任务在单台机器的磁盘和内存上执行。Spark 的任务是协调这些机器为一个整体的计算平台。Spark 是一个分布式的平台，这对于支持任意规模的数据至关重要，而且 Spark 在这一方面做得非常好。它既可以在一台机器上以“本地模式”运行得很好，也可以在数千台机器组成的集群上运行得不错。这符合我们的工具需要支持任意数据规模的要求。Spark 也是优秀的黏合剂，它提供了包括 Kafka 和 MongoDB 等很多系统的连接器，可以把各种系统整合到一起。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

我们在本书第 1 版中使用的是 Hadoop，而 Spark 是对 Hadoop 迭代改进的项目。Spark 快速发展，已经替代 MapReduce 成为 Hadoop 集群上主流的作业系统。Spark 在 Hadoop 分布式文件系统（HDFS）（*http://bit.ly/2oL7OJQ*）和 Apache Hadoop Common（*http://bit.ly/2oCuFa7*）的基础上，把处理过程从磁盘移到了内存来进行加速。在本书这一版中，Spark 完全替代了 Hadoop（见图 2-12）。Spark 要快得多，也相对更好用！

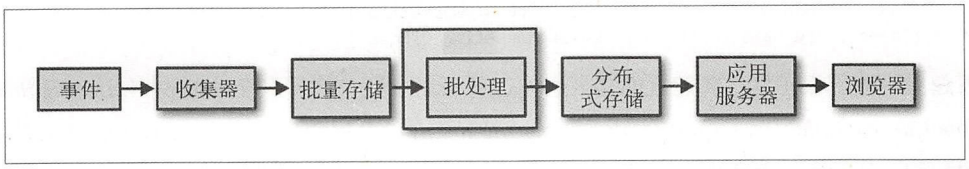


图 2-12　使用 Spark 进行数据处理

需要 Hadoop

Spark 是基于 Hadoop 生态系统构建的，这使它得以迅速崛起。Spark 现在已经在顶级工作团队中很大程度地替代了 Hadoop，而在企业方面还稍显落后。在使用 Spark 之前，我们先简要了解它并安装 Hadoop。在 Vagrant 和 EC2 映像中，这些都已经帮你做好了，如果你要手动安装，请参考附录 A。

用 Spark 处理数据

图 2-13 展示了 Apache Spark 生态系统。Spark 基于 HDFS 或 S3 运行，提供了 Spark SQL、Spark MLlib 和 Spark Streaming 等组件。

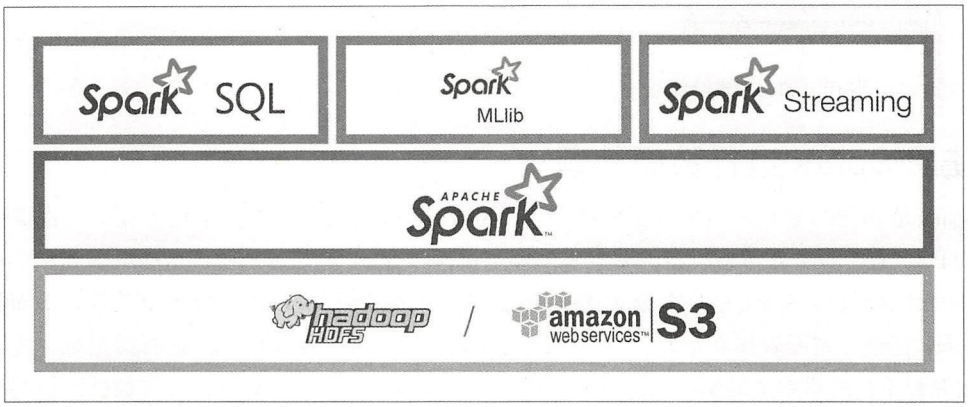


图 2-13　Apache Spark 生态系统

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

Spark 的本地模式可以让我们在开发中在本地的小规模数据集上运行 Spark。在整本书中我们都使用 Spark 本地模式。这样你就可以通过本地开发进行学习，而等数据量变大的时候再迁移到 Spark 集群上。尽管我们注意到随着 EC2 实例可以支持最大 2TB 内存（*http://amzn.to/2oyyI5D*），Spark 依旧可以在本地模式下处理相当大的数据集。我们使用集群而不是这种巨型实例的理由是集群能通过冗余提供更高的可靠性，同时性价比更好，因为多台便宜的机器加起来也没有一台性能怪兽昂贵。

在安装好了 Spark 及其依赖，并配置好环境之后，我们可以简单试用一下 Spark。你可以使用 `pyspark` 命令在任何位置启动 PySpark，不过如果是要运行本书的示例代码，你应该从 *Agile_Data_Code_2* 项目根目录启动 PySpark。如果你是 Spark 新手，不妨阅读 Spark 编程指南（*https://spark.apache.org/docs/1.6.1/programming-guide.html*），然后跟着做一遍。

启动 PySpark 后应该会看到如图 2-14 所示的提示符。

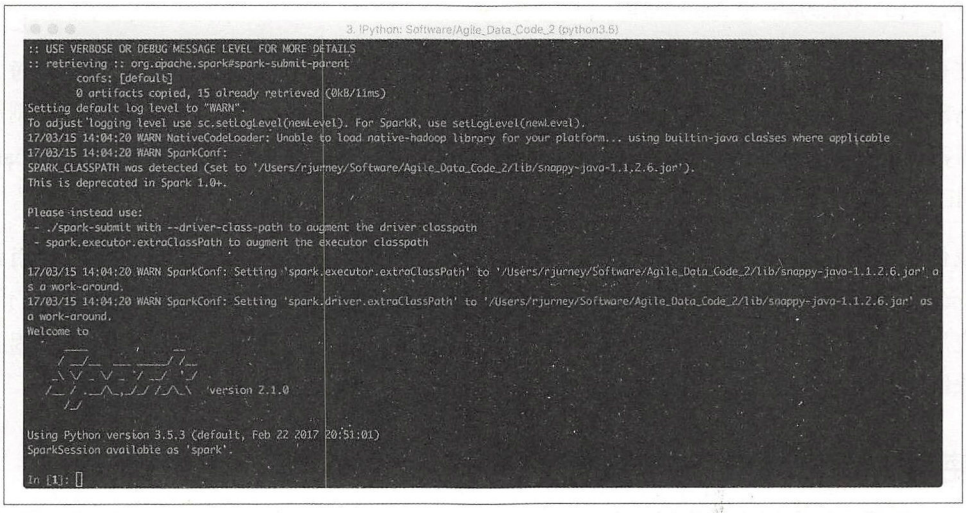


图 2-14　iPython PySpark 控制台

输入下列代码：

```
csv_lines = sc.textFile("data/example.csv")
data = csv_lines.map(lambda line: line.split(","))
data.collect()
```

输出结果如图 2-15 所示。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

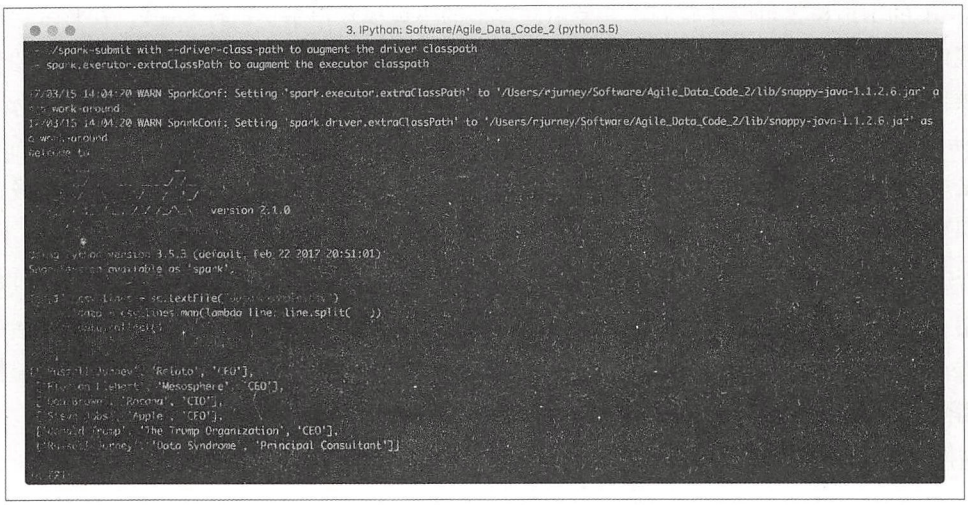


图 2-15　我们的第一个 Spark 程序

在使用 PySpark 时，你最好在浏览器单独的标签页中开着 API 文档，以便临时参考。PySpark 有两套 API：RDD（*http://bit.ly/2p5IM9z*）和 DataFrame。（你可能需要分别在浏览器不同的标签页中查询 RDD、DataFrame 和 MLlib 的相关文档。）你可能还需要查询 Spark ML 文档。

我们已经完成了 Spark 的“Hello, World!”程序！需要处理数据的时候我们会随时用到 Spark。即使现在我们的数据量还不至于需要 Spark，但通过使用 Spark 我们证明了我们的应用程序未来足以应对数据增长和应用伸展。我们的模式是“聚合、处理、发布”，因此我们的数据库不会做比排序更复杂的任务。

使用 MongoDB 发布数据

Spark 不能直接和网络应用服务器通信。为了把数据提供给网络应用，我们需要把它发布到某个数据库中。有很多不错的选择，我们选择的是 MongoDB，因为它简单易用，面向文档，而且和 Spark 集成得很好（见图 2-16）。有了 MongoDB 和 PySpark，我们可以在 PySpark 中定义任意的结构，然后使用 MongoDB 把它保存为与结构相对应的关系。在我们创建新的关系时，不用费心管理表结构，因为我们在 PySpark 中就把数据处理成了用来发布的形式。这就是敏捷！这符合我们对于工具栈各层之间的数据传输只需一条简单命令或者一行代码的要求。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

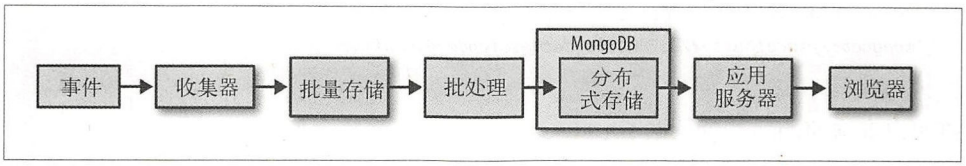


图 2-16　发布数据到 MongoDB

启动 Mongo

要入门 MongoDB，你只需要启动 Mongo 客户端，提供给它数据库名字：

```
mongo agile_data_science
```

这会启动 Mongo 控制台，控制台使用 JavaScript 语言。它会提供一个 db 对象，可以用来与数据库进行交互操作。在 Mongo 中表的概念不称为表，而称为集合。你可以这样向一个集合中插入一个文档：

```
>db.my_collection.insert({"name": "Russell Jurney"});
```

使用如下语句获取一条记录：

```
>db.my_collection.find({"name": "Russell Jurney"});
{ "_id": ObjectId("58cb6959271b8bc38063eb01"), "name": "Russell Jurney"}
```

这就是我们目前对数据库的全部需求，所以暂时就讲到这里。我们使用 Spark 进行数据分析，把数据发布到 Mongo 中。在 Mongo 中要进行的最复杂的数据处理就是列出记录和排序。

从 PySpark 向 MongoDB 推送数据

从 PySpark 向 MongoDB 推送数据是很简单的。

在使用 mongo-hadoop（*https://github.com/mongodb/mongo-hadoop*）项目配置 PySpark 成功连上 MongoDB 之后，我们可以照常使用 PySpark。如 *ch02/pyspark_mongodb.py* 所示，我们可以使用 pymongo_spark 模块把文档存储到刚才用的 MongoDB 中。注意：需要引入并激活 pymongo_spark 包来给 RDD 的接口加上 saveToMongoDB 方法：

```
import pymongo_spark
# 重要：激活 pymongo_spark
pymongo_spark.activate()

csv_lines = sc.textFile("data/example.csv")
data = csv_lines.map(lambda line: line.split(","))
schema_data = data.map(
  lambda x: {'name': x[0], 'company': x[1], 'title': x[2]}
)
```


仅供非商业用途或交流学习使用

```
schema_data.saveToMongoDB(
'mongodb://localhost:27017/agile_data_science.executives'
)
```

现在可以查询 Mongo 中的数据了：

```
$ mongo agile_data_science
$ >db.executives.find()
{ "_id": ObjectId("56f3231cd6ee8112ccbba785"),
  "name": "Don Brown", "company": "Rocana",
  "title": "CIO"}
{ "_id": ObjectId("56f3231cd6ee8112ccbba783"),
  "name": "Russell Jurney", "company": "Relato",
  "title": "CEO"}
{ "_id": ObjectId("56f3231cd6ee8112ccbba784"),
  "name": "Florian Liebert", "company":
  "Mesosphere", "title": "CEO"}
```

恭喜，你已经成功从 Spark 把数据发布到了 NoSQL 数据库中！看看这有多简单：准备好数据之后，只需一行代码就能把数据发布到 Mongo 中。不用担心表结构，这正是我们的工作方式所需要的。在我们准备好要存储之前，我们也不知道表结构会是什么样子的，因此用 PySpark 代码在外部声明表结构几乎没有什么意义。这只是整个软件栈的一部分，但是这个属性确实能帮助我们快速工作，获得敏捷性。

使用 Elasticsearch 搜索数据

Elasticsearch（https://www.elastic.co/cn/）是“用来搜索的 Hadoop”，因为它提供了健壮而易用的搜索解决方案，降低了个人实现数据搜索的门槛，不论数据量是大还是小。Elasticsearch 有一个简单的 RESTful 的 JSON 接口，因此我们可以在命令行中使用任何语言使用它。我们将使用 Elasticsearch 来搜索数据，让我们能轻松实现对费劲创建出的记录的搜索。

Elasticsearch 应该已经在你启动的 Vagrant 或者 EC2 映像中启动了，如果没启动，使用这条命令启动它：

```
elasticsearch -d
```

用 `curl`（*https://curl.haxx.se/*）命令查询 Elasticsearch 很简单。在 Vagrant 和 EC2 映像中 `curl` 已经预装好了，如果你在自己电脑上运行代码，也需要预先装好它。

要在 Elasticsearch 上创建一个索引 `agile_data_science`，你可以使用 `curl`。查阅 Elasticsearch 的索引创建的文档，这个页面有“复制为 curl 命令”的按钮，可以提供各个示例操作的 `curl` 命令。注意：所使用的本地或者云端的 Elasticsearch 守护进程应当监听

仅供非商业用途或交流学习使用

9200 端口才能直接执行。

我们要创建一个分片为 1 且数据镜像为 1 的索引，这对开发来说够用了。如果是生产环境，你可能要让索引有多个分片，同时保留多份数据镜像，以满足数据冗余和性能的要求。你可以在 Vagrant/EC2 映像中运行如下命令：

```
curl -XPUT 'localhost:9200/agile_data_science?pretty' \
  -H 'Content-Type: application/json' -d'
{
    "settings" : {
        "index" : {
            "number_of_shards" : 1,
            "number_of_replicas" : 1
        }
    }
}
'
```

这条命令执行成功会返回如下 JSON 格式的数据：

```
{
"acknowledged": true,
"shards_acknowledged": true
}
```

下面我们尝试把一条文档插入到测试索引中，然后搜索该文档。查阅插入索引的文档。插入命令使用 HTTP PUT 请求：

```
curl -XPUT 'localhost:9200/agile_data_science/test/1?pretty' \
  -H 'Content-Type: application/json' -d'
{
    "name" : "Russell Jurney",
    "message" : "trying out Elasticsearch"
}
'
```

得到如下消息，表示执行成功：

```
{
  "_index": "agile_data_science",
  "_type": "test",
  "_id": "1",
  "_version": 1,
  "result": "created",
  "_shards": {
  "total": 2,
  "successful": 1,
  "failed": 0
  },
  "created": true
}
```

查阅搜索索引的文档。搜索命令使用 HTTP GET 请求：

```
curl -XGET 'localhost:9200/agile_data_science/_search?q=name:Russell&pretty'
```

我们得到了记录、查询执行的描述，以及记录所在的索引信息：

```
{
  "took": 3,
  "timed_out": false,
  "_shards": {
    "total": 1,
    "successful": 1,
    "failed": 0
  },
  "hits": {
    "total": 1,
    "max_score": 0.25811607,
    "hits": [
      {
        "_index": "agile_data_science",
        "_type": "test",
        "_id": "1",
        "_score": 0.25811607,
        "_source": {
          "name": "Russell Jurney",
          "message": "trying out Elasticsearch"
       }
     }
   ]
  }
}
```

关于 Elasticsearch 就暂时讲这么多。下面我们尝试从 PySpark 向 Elasticsearch 中写数据吧！

Elasticsearch 与 PySpark

要把数据从 PySpark 写入 Elasticsearch 中（或者从 Elasticsearch 读取数据到 PySpark 中），我们要使用 Elasticsearch for Hadoop（*https://www.elastic.co/products/hadoop*）。在我准备好的映像中，我们已经为本项目配置好了 PySpark，因此你不用再做什么就可以加载这个库了。如果你是手动安装的，我们也可以通过安装脚本获得类似的配置（见附录 A）。

让 PySpark 数据可以被搜索。我们用 *ch02/pyspark_elasticsearch.py*（*https://github.com/rjurney/Agile_Data_Code_2/blob/master/ch02/pyspark_elasticsearch.py*）把数据从 PySpark 保存到 Elasticsearch 中：

```
csv_lines = sc.textFile("data/example.csv")
data = csv_lines.map(lambda line: line.split(","))
schema_data = data.map(
```

```
    lambda x: ('ignored_key', {'name': x[0], 'company': x[1], 'title': x[2]})
  )
  schema_data.saveAsNewAPIHadoopFile(
    path='-',
    outputFormatClass="org.elasticsearch.hadoop.mr.EsOutputFormat",
    keyClass="org.apache.hadoop.io.NullWritable",
    valueClass="org.elasticsearch.hadoop.mr.LinkedMapWritable",
    conf={ "es.resource": "agile_data_science/executives"})
```

搜索数据。现在，搜索数据就很简单了，我们使用 `curl`：

```
curl \
  'localhost:9200/agile_data_science/executives/_search?q=name:Russell*&pretty'
```

结果如下：

```
{
  "took": 19,
  "timed_out": false,
  "_shards": {
    "total": 1,
    "successful": 1,
    "failed": 0
  },
  "hits": {
    "total": 2,
    "max_score": 1.0,
    "hits": [
      {
        "_index": "agile_data_science",
        "_type": "executives",
        "_id": "AVrfrAbdfdS5Z0IiIt78",
        "_score": 1.0,
        "_source": {
          "company": "Relato",
          "name": "Russell Jurney",
          "title": "CEO"
        }
      },
      {
        "_index": "agile_data_science",
        "_type": "executives",
        "_id": "AVrfrAbdfdS5Z0IiIt79",
        "_score": 1.0,
        "_source": {
          "company": "Data Syndrome",
          "name": "Russell Jurney",
          "title": "Principal Consultant"
        }
      }
    ]
  }
}
```

Elasticsearch 为我们生成了 _id 字段。现在可以告诉大家，Elasticsearch 也是一个很好的键值对存储系统，也就是文档存储系统！它可以轻易替代我们软件栈中的 MongoDB，而且这么做减少了软件栈的组件，能简化并提升软件栈的伸缩性。记住，简单是伸缩性的关键。话虽如此，MongoDB 还有一些我们后面要用到的功能，所以不要不把它放在心上。

通过 pyelasticsearch 使用 Python 操作 Elasticsearch

pyelasticsearch（*http://pyelasticsearch.readthedocs.org/en/latest/*）是从 Python 中访问 Elasticsearch 数据的一种好办法。使用 pyelasticsearch 很简单——运行 *ch02/test_elasticsearch.py*（*http://bit.ly/2oewEPo*）：

```
from pyelasticsearch import ElasticSearch
es = ElasticSearch('http://localhost:9200/')
es.search('name:Russell', index='agile_data_science')
```

结果如下：

```
{'_shards': {'failed': 0, 'successful': 1, 'total': 1},
 'hits': {'hits': [{'_id': '1',
    '_index': 'agile_data_science',
    '_score': 0.7417181,
    '_source': {'message': 'trying out Elasticsearch',
     'name': 'Russell Jurney'},
    '_type': 'test'},
  {'_id': 'AVrfrAbdfdS5Z0IiIt78',
   '_index': 'agile_data_science',
   '_score': 0.7417181,
   '_source': {'company': 'Relato', 'name': 'Russell Jurney', 'title': 'CEO'},
   '_type': 'executives'},
  {'_id': 'AVrfrAbdfdS5Z0IiIt79',
   '_index': 'agile_data_science',
   '_score': 0.7417181,
   '_source': {'company': 'Data Syndrome',
    'name': 'Russell Jurney',
    'title': 'Principal Consultant'},
   '_type': 'executives'}],
  'max_score': 0.7417181,
 'total': 3},
 'timed_out': False,
 'took': 3}
```

用 pyelasticsearch 进行搜索和用 curl 一样简单。

使用 Apache Kafka 分发流数据

Kafka 官网（*https://kafka.apache.org/*）上写道：“Kafka^{TM} 可以用来构建实时数据管道和流计算应用。它可以水平伸缩，具有容错性，且速度极快，在上千家公司的生产环境中运行。”我们要使用 Kafka 流数据和 Spark Streaming 来进行“准实时”的预测。

Kafka 也可以用来收集数据，聚合存入 HDFS 或 Amazon S3 这样的批量存储中。

启动 Kafka

在提供的映像中，Zookeeper 和 Kafka 都已经在运行了。如果没有使用提供的映像文件，你需要在启动 Kafka 之前先启动 Apache Zookeeper（*https://zookeeper.apache.org/*）。Zookeeper 帮助管理 Kafka。为 Zookeeper 启动一个新的控制台，运行：

```
kafka/bin/zookeeper-server-start.sh kafka/config/zookeeper.properties
```

然后，在一个新的控制台中运行 Kafka 服务器：

```
kafka/bin/kafka-server-start.sh kafka/config/server.properties
```

主题，控制台生产者，控制台消费者

Kafka 消息是按主题分组的，所以在我们通过 Kafka 发送消息前，先创建一个主题：

```
$ kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181 \
  --replication-factor 1 --partitions 1 --topic test
Created topic "test".
```

我们可以通过列出主题命令查看我们创建的所有主题：

```
$ kafka/bin/kafka-topics.sh --list --zookeeper localhost:2181
test
```

接下来，我们可以使用“控制台生产者”手动输入一些消息，把它们发送给 test 主题。输入如下命令：

```
kafka/bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test
```

然后输入一条简单的 JSON 消息，按下回车键（由于不会有输出，在输入完成后可以按 Ctrl+C 组合键退出）：

```
{"message": "Hello, World!”}
```

现在，我们可以从头开始回放 test 主题，并查看我们输入的消息。同样，按 Ctrl+C 组合键退出：

```
$ kafka/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 \
  --topic test --from-beginning
{"message": "Hello, World!"}
^CProcessed a total of 1 messages
```

Spark 的实时计算和批量计算

使用 Kafka 很简单，但是我们稍后会看到：这样一个简单的框架可以用简单的方式创建复杂的数据流。Kafka 提供的全局队列的抽象更是非常强大。我们只会用 Kafka 来部署 Spark Streaming 实现的预测，但是 Kafka 其实还有很多其他的功能。

尽管 Kafka 很强大，但是我们会在本书中花大部分时间来做批量计算。我们的原则是："只要能用批量模式，就用批量模式。"运维一个 Spark 集群比运维一堆 Kafka 实时工作者要容易得多。尽管你可以回放 Kafka 的历史消息来实现和批量操作等价的效果，但批量计算才是专为组成数据科学的应用研究过程而优化的。

如果你决定从批量计算转到实时流计算，PySpark 还是能满足你的需求的！在 PySpark 批量计算模式中用来处理消息的代码，可以原封不动地放到 PySpark Streaming 中来处理来自 Kafka 的消息。在批量计算模式下实现流处理应用的原型，然后转为流计算模式，就非常自然了。

通过 kafka-python 实现来在 Python 中使用 Kafka

kafka-python（*https://github.com/dpkp/kafka-python*）为使用 Python 操作 Kafka 提供了一种简易方式。让我们打开 Python 控制台，写一个简单的程序来从我们刚刚创建的 `test` 主题中读取数据，以此试用一下 kafka-python。你可以照着 ch02/python_kafka.py 做，同时阅读 KafkaConsumer 文档（*http://bit.ly/2nPvu0x*）。创建消费者只需要一行代码，但是要定位到一个主题的消息的开头，我们需要把消费者分配到分区 0。然后调用 `seek_to_beginning` 并使用消费者遍历读取各条消息。

注意：我们的消息是以字节码形式存在的，所以我们必须调用 `bytes.decode`（*https://docs.python.org/3/library/stdtypes.html#bytes.decode*）对它进行解码，然后再让 JSON 解析（如果使用 Python 2 就不需要了）：

```
import sys, os, re
import json

from kafka import KafkaConsumer, TopicPartition
consumer = KafkaConsumer()
consumer.assign([TopicPartition('test', 0)])
consumer.seek_to_beginning()

for message in consumer:
message_bytes = message.value
message_string = message_bytes.decode()
message_object = json.loads(message_string)
print(message_object)
```

仅供非商业用途或交流学习使用

输出如下：

```
{'message': 'Hello, World!'}
```

即使是在这条消息打印出来之后，循环也依然在继续。这是正常情况下会发生的，所以你需要按下 Ctrl+C 组合键退出循环。

这就是 Kafka！我们会在第 8 章中使用 `kafka-python` 包来从 Flask 网络应用中发出预测事件，然后在 PySpark Streaming 中完成处理。我们将会使用 PySpark Streaming 来处理大规模的 Kafka 流数据中的消息。

直接继续，让 Zookeeper 和 Kafka 先继续运行一段时间，因为我们在下一节中还要用到它们。

使用 PySpark Streaming 处理流数据

使用 PySpark Streaming 处理 Kafka 流数据要比使用普通的 Spark 稍微复杂一些。首先，在另一个 SSH 控制台中启动一个控制台生产者，让它空闲运转一会：

```
kafka/bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test
```

然后，把工作目录切换到 *Agile_Data_Code_2* 目录下。要运行 PySpark Streaming，你需要把 `spark-streaming-kafka` 的 Maven 包添加到命令行参数里：

```
pyspark --packages org.apache.spark:spark-streaming-kafka-0-8_2.11:2.1.0
```

现在，使用 iPython 执行下面的代码，可以初始化一个 PySpark 的 `StreamingContext`。你可以使用 *ch02/pyspark_streaming.py* 跟着做。注意：`PERIOD` 定义了 Spark Streaming 多久进行一次迷你批处理——在本例中，每 10 秒钟会触发一次：

```
import sys, os, re
import json

from pyspark import SparkContext, SparkConf
from pyspark.streaming import StreamingContext
from pyspark.streaming.kafka import KafkaUtils, OffsetRange, TopicAndPartition

# 每 10 秒钟处理一次数据
PERIOD=10
BROKERS='localhost:9092'
TOPIC='test'

conf = SparkConf().set("spark.default.parallelism", 1)
sc = SparkContext(
  appName = "Agile Data Science: PySpark Streaming 'Hello, World!'", conf=conf
)
ssc = StreamingContext(sc, PERIOD)
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

有了 StreamingContext，我们就可以创建一个 Kafka 流了：

```
stream = KafkaUtils.createDirectStream(
  ssc,
  [TOPIC], {
    "metadata.broker.list": BROKERS,
    "group.id": "0",
  }
)
```

最后，我们可以读取 JSON 格式的消息并打印到控制台里：

```
object_stream = stream.map(lambda x: json.loads(x[1]))
object_stream.pprint()
```

要启动 StreamingContext 并开始处理 Kafka 消息，只要运行：

```
ssc.start()
```

现在，在之前启动的 Kafka 的控制台生产者的框框里面输入一条简单的 JSON 消息，然后按回车键：

```
{"message": "Testing PySpark Streaming!"}
```

切回 iPython 控制台，10 秒钟以内我们就会看到如下的输出：

```
-------------------------------------------
Time: 2016-11-19 19:54:50
-------------------------------------------
{'message': 'Testing PySpark Streaming'}
```

这就是使用 PySpark Streaming 处理 Kafka 流的方法！我们会在第 8 章中再次用到 Spark Streaming，来把 Spark MLlib 分类器部署为实时运行的模式。你可以暂时先把 Zookeeper 和 Kafka 的控制台关掉，别忘了把控制台生产者也关掉。

使用 scikit-learn 与 Spark MLlib 进行机器学习

我们要使用 scikit-learn（简称 sklearn）和 Spark MLlib 构建预测模型。我们会使用 sklearn 做回归分析，使用 Spark MLlib 做分类。

为什么有了 Spark MLlib 还要使用 scikit-learn

尽管 Spark 通过 Spark MLlib（*http://spark.apache.org/mllib/*）提供了机器学习的功能，但 scikit-learn 也包含了许多 MLlib 所缺失的关于数据流和流程的工具。sklearn 还能让我们对新的样本实时进行分类或者回归，而不需要 Kafka 和 Spark Streaming，这样会简单很多。

本书中除 scikit-learn 外使用的都是“大数据”工具，而在这样一本书中引入 scikit-

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

learn 的主要原因是在实践中它真的非常好用。Spark MLlib 是专为大规模数据设计的，而大数据却经常会在提取特征时整合归约为很小的数据集。这意味着 sklearn 有的时候比 Spark MLlib 更好。如果你要在数据流的中间使用简单的机器学习算法，那么务必使用 MLlib。但是如果要实时预测而且数据能放进内存中，请考虑使用 sklearn。我们会在第 7 章中同时用到这两个工具，而在第 8 章和第 9 章中只使用 Spark MLlib。

使用 Apache Airflow（孵化项目）进行调度

Apache Airflow（孵化项目）（*https://airflow.incubator.apache.org/*）是有向无环图（DAG）的一种调度器。我们将在 PySpark 中创建一些数据管道，而 DAG 对于描述这种数据管道非常方便。Airflow 让我们把长长的数据管道分为多个逻辑上相连的脚本。我们将使用 Airflow 部署构成预测应用的数据管道（也就是“数据流”）。Airflow 让我们可以调度应用来定期执行，比如每天、每小时等。

Airflow 正在成为领先的开源数据管道调度器，因为它使用 Python 代码进行控制，而不是使用配置文件。这是配置调度器的一种更“干净”的方式。

Airflow 是一个批量计算的工具。值得注意的是，只要能把应用部署为批处理模式，可能就应该把它部署为批处理模式。如果能安排应用程序的代码每天或者每小时甚至是每 10 分钟运行一次，那么部署、运维和维护的工作就简单得多了。使用调度器按时运行任务的操作比使用实时系统持续运行的操作要简单很多（尽管随着 Kafka 发展成熟，这种实时系统已经没那么难了）。

Airflow 的代码可以在 GitHub（*https://github.com/apache/incubator-airflow*）上找到。Airbnb 创建了 Airflow 项目，并且提供了一个带有截图、视频，以及其他文档的精美的 Airflow 介绍页面（*http://nerds.airbnb.com/airflow/*）。注意，我们是以开发为目的配置 Airflow 的。要在生产环境中使用的话，你需要先验证 Airflow 是不是能和真正的 Spark 集群共同工作。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

小心使用 Oozie

应用程序以批处理和实时处理的模式进行部署哪个更简单，在很大程度上取决于调度器的选择。有些系统使用了粗暴的设定和冗长的配置，而那些使用习惯与之截然相反的系统要更容易运营。

虽然 Apache Oozie 是领先的 Hadoop 发行版中使用的标准调度器，**受迫于截止时间的项目要使用它最好还是格外小心**。在我工作过的一家初创企业里，我们计划过专门分配一个完整的人力来为单个应用运维 Apache Oozie。奉劝各位：仅仅因为在所使用的 Hadoop 或者 Spark 发行版中已经包含了 Oozie 就准备使用它之前，先调研一下像 Azkaban 和 Apache Airflow 这种更符合正常使用惯例的系统。

Oozie 动辄需要篇幅长达好几页的 XML 代码来实现非常简单的任务。和真正的编程语言比起来，图灵完备的 XML 语言是程序员的噩梦。Oozie 是针对最复杂的大企业里最复杂的应用而优化的。如果你的公司和项目并不是这样的，那么最好和 Oozie 保持距离。将来你会庆幸这么做是对的。这不是对 Oozie 开发者的人身攻击，他们是为了满足雅虎和其他大公司苛刻的企业调度需求而开发它的，付出的代价则是对于常见任务没有那么好用。

Oozie 的替代品包括 Azkaban（*https://azkaban.github.io/*）、Luigi（*https://github.com/spotify/luigi*），以及 Apache Airflow（*https://airflow. incubator.apache.org/*）。在使用 Oozie 之前不妨先试试这些项目。

安装 Airflow

Airflow 是通过 `pip` 安装的，在提供的 Vagrant/EC2 映像上已经预装好了。你可以照着 Airflow 安装指南和 Airflow 配置指南来做。Airflow 只是一个 `pip` 模块，所以不管在哪里安装都很简单。Airflow 还提供了很多附加的包以供选装，比如：如果你要安装 MySQL 或者 Postgres 支持，那么请参阅安装指南的附加包这一小节。

我们使用 `airflow` 命令与 Airflow 交互，控制 Airflow 调度器和网络应用。Airflow 存放数据库、配置文件和 DAG 的默认路径是 `~/airflow/`：

```
$ ls ~/airflow
dags  logs  plugins
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

现在，访问 Airflow 的网页界面 *http://localhost:8080/admin/*。你会看到一个和图 2-17 类似的界面。

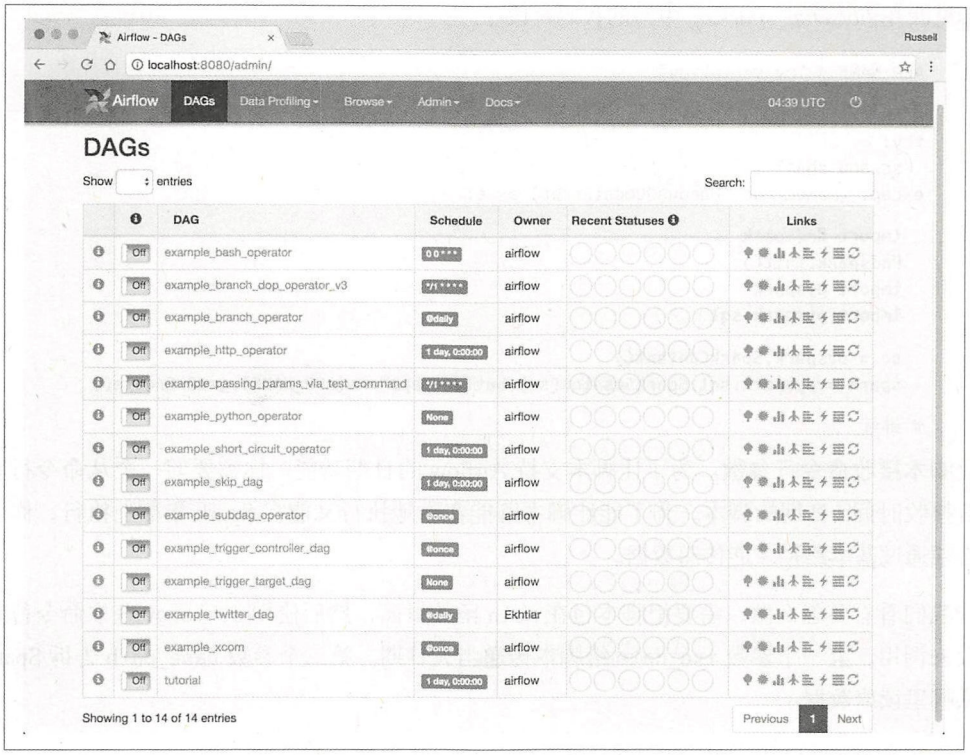


图 2-17　Airflow 网页界面

准备给 Airflow 用的脚本

尽管文档中没有说，要创建可以同时在本地和生产环境中用于 Airflow 和 Spark 集群的脚本还需要一些工作。首先，我们需要有条件地配置 PySpark 环境。接下来，我们要参数化我们的脚本，这样我们可以在 `bash` 中调用命令行，并传递日期和数据的相对路径。这几件事都完成后，我们可以得到既能用于开发环境的 PySpark 控制台，又能用于生产环境中的 Airflow 的脚本。

有条件地初始化 PySpark。有一种简单的方法可以让我们写出的 Spark 脚本既能在 PySpark 控制台中交互使用，又能给 Airflow 通过 `spark-submit` 提交到集群上执行。我们使用 Python 包 `findspark`（*https://github.com/minrk/findspark*）来有条件地创建 PySpark 脚本（包括本书中的脚本）中使用的 Spark 环境和会话对象，仅当它们不存在时创建，因为在使用

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

PySpark 控制台时这些变量也许已经被创建出来了。这样，脚本就既能在 PySpark 控制台中执行，又能通过 spark-submit 执行了。

参见我在 *lib/setup_spark.py* 中写的代码片段：

```
APP_NAME = "my_script.py"

# 如果没有 SparkSession 的话，则准备好环境
try:
  sc and spark
except (NameError, UnboundLocalError) as e:

  import findspark
  findspark.init()
  import pyspark
  import pyspark.sql

  sc = pyspark.SparkContext()
  spark = pyspark.sql.SparkSession(sc).builder.appName(APP_NAME).getOrCreate()

# 继续 ...
```

让脚本接收命令行参数。为了让脚本支持 Airflow 的日期功能，你需要写一个从命令行参数接收时间 / 日期的脚本。为了能让脚本既能在本地执行又能在 Spark 集群上执行，你还必须通过基本路径才能传递数据。

让我们看看要怎么做。需要把脚本包在 main 函数里面，然后使用 sys.argv 获取命令行参数来调用。第一个参数 iso_date 给脚本传递当天日期。第二个参数 base_path 告诉 Spark 从哪里读取数据：

```
# 从 Airflow 中向 main() 传递日期和基本路径
def main(iso_date, base_path):
  APP_NAME = "pyspark_task_one.py"

  ...

  # 获取今天的日期
  today_dt = iso8601.parse_date(iso_date)
  rounded_today = today_dt.date()

  # 读取今天的日期
  today_input_path = "{}/ch02/data/example_name_titles_daily.json/{}".format(
    base_path,
    rounded_today.isoformat()
  )

...

if __name__ == "__main__":
  main(sys.argv[1], sys.argv[2])
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

这样，这个脚本可以使用一行命令执行：

```
python ch02/pyspark_task_one.py 2016-12-01 .
```

用 Python 创建 Airflow DAG

让我们尝试使用 Airflow 运行一个简单的任务。记住：不要把文件命名为 *airflow.py*，否则会和 Airflow 项目本身的 Python 引入发生混淆！

我们要做的第一件事就是初始化 Airflow 数据库，如果还没初始化过的话：

```
airflow initdb
```

接下来，我们要为我们的 DAG 配置脚本 *airflow_test.py* 在 Airflow 的 DAG 目录 *~/airflow/dags* 中创建软链接。如果不在 *~/airflow/dags* 目录中是没法工作的。如 *ch02/setup_airflow_test.sh* 所示：

```
#!/usr/bin/env bash

ln -s $PROJECT_HOME/ch02/airflow_setup.py ~/airflow/dags/
```

我们的配置脚本 *airflow_test.py* 也比较简单。首先我们定义了一个配置项对象，然后用它来创建 DAG：

```
import sys, os, re

from airflow import DAG
from airflow.operators.bash_operator import BashOperator

from datetime import datetime, timedelta
import iso8601

project_home = os.environ["PROJECT_HOME"]

default_args = {
  'owner': 'airflow',
  'depends_on_past': False,
  'start_date': iso8601.parse_date("2016-12-01"),
  'email': ['russell.jurney@gmail.com'],
  'email_on_failure': True,
  'email_on_retry': True,
  'retries': 3,
  'retry_delay': timedelta(minutes=5),
}

# timedelta 1 表示每天执行
dag = DAG(
  'agile_data_science_airflow_test',
  default_args=default_args,
  schedule_interval=timedelta(1)
)
```

仅供非商业用途或交流学习使用

接下来，为数据流中的每个脚本创建一个 BashOperator（*https://airflow.incubator.apache.org/code.html#airflow.operators.BashOperator*）。定义从 bash 运行我们脚本的命令，以及所用的一些参数和路径，然后使用内建参数和刚才定义的参数把命令补充完整。ds 变量是一个内建变量，包含 Airflow 运行该命令的日期。我们还提供了脚本要用的文件名 filename 以及基本路径 base_path：

```
# 运行一个简单的 PySpark 脚本
pyspark_local_task_one = BashOperator(
  task_id = "pyspark_local_task_one",
  bash_command = """spark-submit \
  --master {{ params.master }}
  {{ params.base_path }}/{{ params.filename }} {{ ds }} {{ params.base_path }}
  """,
  params = {
    "master": "local[8]",
    "filename": "ch02/pyspark_task_one.py",
    "base_path": "{}/".format(project_home)
},
dag=dag
)

# 运行另一个简单的 PySpark 脚本，依赖前一个脚本
pyspark_local_task_two = BashOperator(
  task_id = "pyspark_local_task_two",
  bash_command = """spark-submit \
  --master {{ params.master }}
  {{ params.base_path }}/{{ params.filename }} {{ ds }} {{ params.base_path }}
  """,
  params = {
    "master": "local[8]",
    "filename": "ch02/pyspark_task_two.py",
    "base_path": "{}/".format(project_home)
  },
  dag=dag
)
```

最后，设置第一个脚本和第二个脚本之间的依赖关系：

```
# 添加从第二个任务到第一个任务的依赖关系
pyspark_local_task_two.set_upstream(pyspark_local_task_one)
```

现在，我们只要运行在 *~/airflow/dags* 中创建了软链接的脚本，它就会出现在 Airflow 系统中了。注意：脚本必须要在 *~/airflow/dags* 中创建文件链接或者复制进去，否则运行它不会有任何作用。还要注意：此处和后面的输出中的时间和日期因为受限于页面宽度而被删掉了：

```
$ python ~/airflow/dags/airflow_test.py
[... 15:04:37,875] {__init__.py:36} INFO - Using executor SequentialExecutor
```

仅供非商业用途或交流学习使用

就是这样！该脚本已经在 Airflow 中创建了一个可以执行、调度、回填的 DAG。让我们看看我们在本例刚刚创建的 DAG 中使用的两个脚本的完整内容。

Airflow 使用的完整脚本

我们创建了两个用于 Airflow DAG 的脚本，它们分别是 *ch02/pyspark_task_one.py* 和 *ch02/pyspark_task_two.py*。这两个脚本是简短的。组合使用时，它们读取名字和头衔的列表，计算出每个名字的完整头衔，然后把结果存入 MongoDB。这两个脚本随 Airflow 的 DAG 被设置为每天运行一次，读入一天的输入数据，写出一天的输出数据。

查看 *ch02/pyspark_task_one.py*，它从今天的输入路径中读取数据，为每个名字创建完整的头衔，然后把结果保存到今天的输出路径中。注意脚本要有 +x 权限才能被 Airflow 执行：

```
#!/usr/bin/env python

import sys, os, re
import json
import datetime, iso8601

# 从 Airflow 中向 main() 传递日期和基础路径
def main(iso_date, base_path):
  APP_NAME = "pyspark_task_one.py"

  # 如果没有 SparkSession 的话，则准备好环境
  try:
    sc and spark
  except NameError as e:
    import findspark
    findspark.init()
    import pyspark
    import pyspark.sql

    sc = pyspark.SparkContext()
    spark = pyspark.sql.SparkSession(sc).builder.appName(APP_NAME).getOrCreate()

  # 获取今天的日期
  today_dt = iso8601.parse_date(iso_date)
  rounded_today = today_dt.date()

  # 读取今天的日期
  today_input_path = "{}/ch02/data/example_name_titles_daily.json/{}".format(
    base_path,
    rounded_today.isoformat()
  )

  # 读取数据并继续 ...
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

```
    people_titles = spark.read.json(today_input_path)
    people_titles.show()

    # 以 RDD 进行分组
    titles_by_name = people_titles.rdd.groupBy(lambda x: x["name"])

    # 接收分组后的数据，把各种头衔连接成完整头衔
    def concatenate_titles(people_titles):
      name = people_titles[0]
      title_records = people_titles[1]
      master_title = ""
      for title_record in sorted(title_records):
        title = title_record["title"]
        master_title += "{}, ".format(title)
      master_title = master_title[:-2]
      record = {"name": name, "master_title": master_title}
      return record

    people_with_contactenated_titles = titles_by_name.map(concatenate_titles)
    people_output_json = people_with_contactenated_titles.map(json.dumps)

    # 获取今天的输出路径
    today_output_path = "{}/ch02/data/example_master_titles_daily.json/{}".format(
      base_path,
      rounded_today.isoformat()
    )

    # 写入 / 覆盖今天的输出路径
    os.system("rm -rf {}".format(today_output_path))
    people_output_json.saveAsTextFile(today_output_path)

  if __name__ == "__main__":
    main(sys.argv[1], sys.argv[2])
```

我们可以使用如下命令测试这个脚本：

```
python ch02/pyspark_task_one.py 2016-12-01 .
```

可以从 debug 输出中看到：

```
+--------------+--------------+
|          name|         title|
+--------------+--------------+
|Russell Jurney|Data Scientist|
|Russell Jurney|        Author|
|Russell Jurney|     Dog Lover|
|     Bob Jones|           CEO|
|     Susan Shu|      Attorney|
+--------------+--------------+
```

仅供非商业用途或交流学习使用

第二个脚本 *ch02/pyspark_task_two.py* 也是类似的，读取第一个脚本的输出，然后把它存储到 MongoDB 中（同样地，这个脚本必须有 +x 权限才能被 Airflow 执行）：

```
#!/usr/bin/env python

import sys, os, re
import json
import datetime, iso8601

# 从 Airflow 中向 main() 传递日期和基础路径
def main(iso_date, base_path):
  APP_NAME = "pyspark_task_two.py"

  # 如果没有 SparkSession 的话，则准备好环境
  try:
    sc and spark
  except NameError as e:
    import findspark
    findspark.init()
    import pyspark
    import pyspark.sql

    sc = pyspark.SparkContext()
    spark = pyspark.sql.SparkSession(sc).builder.appName(APP_NAME).getOrCreate()

  import pymongo
  import pymongo_spark
  # 重要：激活 pymongo_spark
  pymongo_spark.activate()

  # 获取今天的日期
  today_dt = iso8601.parse_date(iso_date)
  rounded_today = today_dt.date()

  # 读取今天的日期
  today_input_path = "{}/ch02/data/example_master_titles_daily.json/{}".format(
    base_path,
    rounded_today.isoformat()
  )

  # 读取数据并继续
  people_master_titles_raw = sc.textFile(today_input_path)
  people_master_titles = people_master_titles_raw.map(json.loads)
  print(people_master_titles.first())

  people_master_titles.saveToMongoDB(
  'mongodb://localhost:27017/agile_data_science.people_master_titles'
  )

if __name__ == "__main__":
  main(sys.argv[1], sys.argv[2])
```

仅供非商业用途或交流学习使用

我们可以使用如下命令测试这个脚本：

```
python ch02/pyspark_task_two.py 2016-12-01 .
```

在 Spark 的输出中，脚本打印了如下的 debug 级日志：

```
{'master_title': 'Author, Data Scientist, Dog Lover', 'name': 'Russell Jurney'}
```

注意：这两个脚本在 DAG 中逻辑相连，这比通过类似 `cron` 的工具来调度它们要简单得多。

在 Airflow 中测试任务

现在我们有了一个 AirflowDAG 和相应的任务，需要通过 Airflow 测试我们的任务。在开始之前，让我们先检查 Airflow 的命令列表：

```
$ airflow

[...,293] {__init__.py:36} INFO - Using executor SequentialExecutor
usage: airflow [-h]
               {variables,worker,upgradedb,task_state,trigger_dag,clear,
               scheduler,resetdb,pause,serve_logs,render,backfill,
               flower,webserver,kerberos,version,list_tasks,
               initdb,list_dags,test,run,unpause}
               ...
airflow: error: the following arguments are required: subcommand
```

让我们先列出所有可用的 DAG，看看我们定义的那个在不在：

```
$ airflow list_dags

agile_data_science_airflow_test
example_bash_operator
example_branch_dop_operator_v3
example_branch_operator
...
```

接下来，列出我们定义的 DAG 中所有的任务：

```
$ airflow list_tasks agile_data_science_airflow_test
pyspark_local_task_one
pyspark_local_task_two
```

现在让我们运行 `pyspark_local_task_one`：

```
airflow test agile_data_science_airflow_test pyspark_local_task_one 2016-12-01
```

我们应该会看到和采用命令行测试 pyspark_task_one.py 脚本同样的输出，尽管我们是通过 Airflow 的 `BashOperator` 运行的：

仅供非商业用途或交流学习使用

```
[...,508] {bash_operator.py:77} INFO - +--------------+--------------+
[...,508] {bash_operator.py:77} INFO - |          name|         title|
[...,508] {bash_operator.py:77} INFO - +--------------+--------------+
[...,508] {bash_operator.py:77} INFO - |Russell Jurney|Data Scientist|
[...,508] {bash_operator.py:77} INFO - |Russell Jurney|        Author|
[...,508] {bash_operator.py:77} INFO - |Russell Jurney|     Dog Lover|
[...,508] {bash_operator.py:77} INFO - |     Bob Jones|           CEO|
[...,508] {bash_operator.py:77} INFO - |     Susan Shu|      Attorney|
[...,508] {bash_operator.py:77} INFO - +--------------+--------------+
[...,508] {bash_operator.py:77} INFO -
[...,953] {bash_operator.py:80} INFO - Command exited with return code 0
```

现在让我们测试一下 pyspark_local_task_two：

```
airflow test agile_data_science_airflow_test pyspark_local_task_two 2016-12-01
```

同样，我们会看到预期的 debug 输出通过 BashOperator 打印出来：

```
[...,046] {bash_operator.py:77} INFO - {'name': 'Russell Jurney', 'master_title':
        'Author, Data Scientist, Dog Lover'}
[...,476] {bash_operator.py:80} INFO - Command exited with return code 0
```

用 Airflow 执行 DAG

现在我们已经分别测试了两个任务，需要让执行结果保存到数据库中，以不可重复的方式执行任务。run 命令的用法和 test 完全一样：

```
airflow run agile_data_science_airflow_test pyspark_local_task_one 2016-12-01
```

你可以在 *~/airflow/logs* 中看到本次执行的日志：

```
$ cat ~/airflow/logs/agile_data_science_airflow_test/pyspark_local_task_one \
        /2016-12-01T00\:00\:00

...

[...,723] {sequential_executor.py:26} INFO - Executing command:
airflow run agile_data_science_airflow_test pyspark_local_task_one
...T00:00:00 --local -sd DAGS_FOLDER/airflow_test.py
[... 15:40:13,815] {models.py:154} INFO - Filling up the DagBag
from /Users/rjurney/airflow/dags/airflow_test.py
[... 15:40:14,951] {models.py:154} INFO - Filling up the DagBag
from /Users/rjurney/airflow/dags/airflow_test.py
[... 15:40:14,997] {models.py:1150} INFO - Task <TaskInstance:
agile_data_science_airflow_test.pyspark_local_task_one 2016-12-01
00:00:00 [success]> previously succeeded on 2016-12-04 15:36:47
.869543
```

使用 clear 命令清除本次的执行记录：

```
airflow clear -s 2016-12-01 -e 2016-12-01 agile_data_science_airflow_test
```

用 Airflow 回填数据

能调度一些操作很不错，但是能不能重做昨天的任务呢？例如，如果我们创建了一种新的预测，除了让它从现在开始每晚执行，我们还需要退回到两周前，拿过去两周的数据进行计算，要怎么做呢？ `backfill` 命令就是处理这种操作的。

只需一行命令就可以回填某一天的数据（我们只有这一天的数据）：

```
airflow backfill -s 2016-12-01 -e 2016-12-01 agile_data_science_airflow_test
```

很酷！这是一个很强大的功能。比如，如果服务器发生故障，则这条命令可以在短时间内轻松重新生成结果。Airflow 让我们不用构建自己的系统就可以应对这种无法避免的情况。

Airflow 的强大作用

我希望本节展示了 Airflow 的强大作用，也展示了为什么我们要费这么大的劲把脚本改为命令的形式来让它们支持 Airflow 的日期处理和 `spark-submit` 的相对路径。把任何 PySpark 脚本迁移到生产环境中都要经历类似的过程，在做这些事的时候，要牢记本节的内容。文档没有说明怎样才能让 Airflow 使用 PySpark，因此把本节内容作为手头的参考吧。

我们还会在第 8 章中进一步讨论 Airflow，到时我们要用 Airflow 以批处理模式部署 PySpark 数据管道。

反思我们的工作流程

和直接从 MySQL 或者 MongoDB 中查询比起来，我们的工作流程看起来更难。但是要注意到，我们的软件栈是为耗时的有深度的数据处理和不时的数据发布而优化的。而且，这样我们就不会在实时查询中越来越复杂，最终导致在无法扩容的时候撞上天花板。

在我们的应用高效地连通之后，我们的团队就可以一起高效工作了，而这在之前是无法实现的。这个软件栈是我们保持敏捷性的基石。

轻量级网络应用

下一步是把我们发布的数据转为可交互的应用。见图 2-18，我们将使用轻量级网站开发框架来实现。

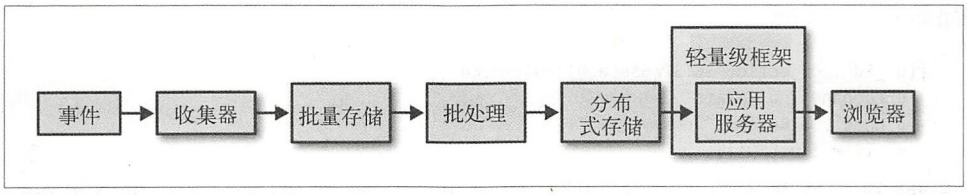


图 2-18　使用 Python 和 Flask 转化为网页

我们选择轻量级网站框架是因为它们简单，用起来很快。和那些 CRUD 不同，挖掘过的数据是我们的主角。我们使用只读的数据库和简单的应用框架，因为它们足以满足我们所构建的应用的要求，也适合于我们提供价值的方式。

有了下面这个使用 Python/Flask 的实例，要使用 Sinatra、Rails、Django、Node.js，或者其他编程语言和网站开发框架也很容易。

Python 与 Flask

Flask 是 Python 的一个快速（*http://bottlepy.org/docs/dev/*）、简单、轻量级的 WSGI 微网站框架。[1]

Flask 官网（*http://flask.pocoo.org/*）提供了一份优秀的 Flask 使用说明。

Flask 回显微服务。运行 Flask 的回显程序 *ch02/web/test_flask.py*：

```
from flask import Flask
app = Flask(__name__)

@app.route("/<input>")
def hello(input):
  return input

if __name__ == "__main__":app.run(debug=True)
```

使用 `curl` 验证是否成功：

```
$ curl http://localhost:5000/hello%20world!
hello world!
```

用 pymongo 在 Python 中使用 Mongo。`pymongo` 是 MongoDB 在 Python 中的一个简单的接口。运行 *ch02/test_pymongo.py* 来试用一下：

```
from pymongo import MongoClient
client = MongoClient()
db = client.agile_data_science
list(db.executives.find({"name": "Russell Jurney"}))
```

1 此处原书引用 Bottle 的文档 (http://bottlepy.org/docs/dev/)，Bottle 也是一个 Python 的轻量级网站开发框架，和 Flask 类似。作者引用的这句话原来的主语为 Bottle，此处把它改为了 Flask。——译者注

输出如下：

```
[{u'_id': ObjectId('56f32e65d6ee81199682dcce'),
  u'company': u'Relato',
  u'name': u'Russell Jurney',
  u'title': u'CEO'}]
```

用 Flask 展示行政人员。现在我们可以使用 pymongo 和 Flask 来展示我们用 Pig 和 MongoStorage 存储到 Mongo 中的 sent_counts 了。运行 *ch02/web/flask_pymongo.py*（*http://bit.ly/2oLplC2*）：

```
from flask import Flask
from pymongo import MongoClient import bson.json_util

# 配置 Flask
app = Flask(__name__)

# 配置 Mongo
client = MongoClient()# 默认连接 localhost
db = client.agile_data_science

# 获取给定 email 地址的全部记录
@app.route("/executive/<name>") def executive(name):
executive = db.executives.find({"name": name})
return bson.json_util.dumps(list(executive))

if __name__ == "__main__": app.run(debug=True)
```

现在用浏览器访问对应的 URL，或者用 curl 命令访问这个服务，就可以看到数据了：

```
[{"company": "Relato",
  "_id": {"$oid": "56f32e65d6ee81199682dcce"},
  "name": "Russell Jurney", "title": "CEO"
}]
```

这一步就这样完成了！（见图 2-19）

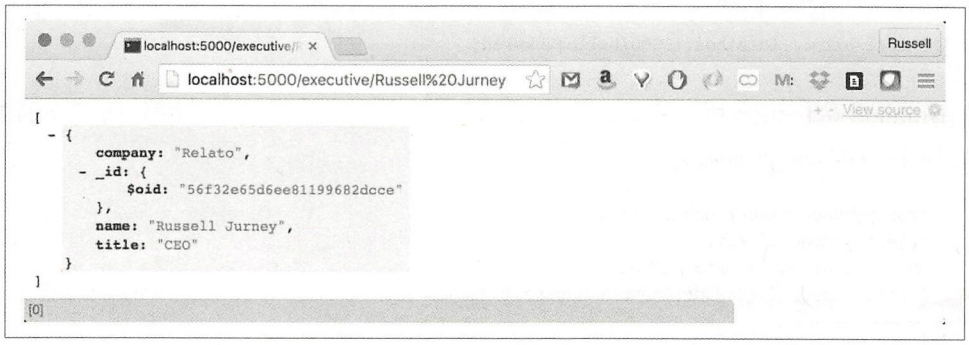


图 2-19　用不加修饰的网页展示数据

仅供非商业用途或交流学习使用

恭喜！数据已经发布到了网页上。现在，让我们把这个页面做得好看一点。

展示数据

设计和展示对于工作成果的价值很重要。事实上，对敏捷数据科学的一种理解就是迭代的数据设计。数据模型的输出是和视图相匹配的，在这种意义上，设计和数据处理没有什么不同。相反，它们是同一个协作行为——数据设计的组成部分。考虑到这一点，我们最好能一开始就有一套稳固而整洁的设计，并基于这套设计开展工作（见图 2-20）。

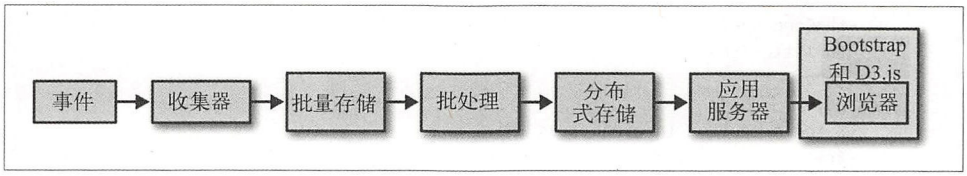


图 2-20　使用 Bootstrap 与 D3.js 展示数据

启动 Bootstrap

让我们尝试使用 Bootstrap 的样式把刚才的例子做成表格。

如 *ch02/web/test_flask_bootstrap.py*（*https://github.com/rjurney/Agile_Data_Code_2/blob/master/ch02/web/test_flask_bootstrap.py*）所示：

```
from flask import Flask, render_template
from pymongo import MongoClient
import bson.json_util

# 配置 Flask
app = Flask(__name__)

# 配置 Mongo
client = MongoClient() # 默认连接 localhost
db = client.agile_data_science

# 获取给定 email 地址的全部记录
@app.route("/executive/<name>")
def executive(name):
  executives = db.executives.find({"name": name})
  return render_template('table.html', executives=list(executives))

if __name__ == "__main__": app.run(debug=True)
```

表格，我的天！

用表格来展示表格状的数据，这就对了！ Bootstrap 让我们可以大胆使用表格。我们将更新控制器来隐藏我们的数据，并且创建一个简单模板把数据输出到表格中。

仅供非商业用途或交流学习使用

然后模板如 *ch02/web/templates/table.html*（*https://github.com/rjurney/Agile_Data_Code_2/blob/master/ch02/web/templates/table.html*）所示：

```
<div class="container">
  <div class="page-header">
    <h1>Agile Data Science</h1>
  </div>
  <p class="lead">Executives</p>
  <table class="table">
    <thead>
      <th>Name</th>
      <th>Company</th>
      <th>Title</th>
    </thead>
    <tbody>
      {% for executive in executives -%}
      <tr>
        <td>{{executive.name}}</td>
        <td>{{executive.company}}</td>
        <td>{{executive.title}}</td>
      </tr>
      {% endfor -%}
    </tbody>
  </table>
</div>
```

结果具有高可读性，见图 2-21，很容易看明白！

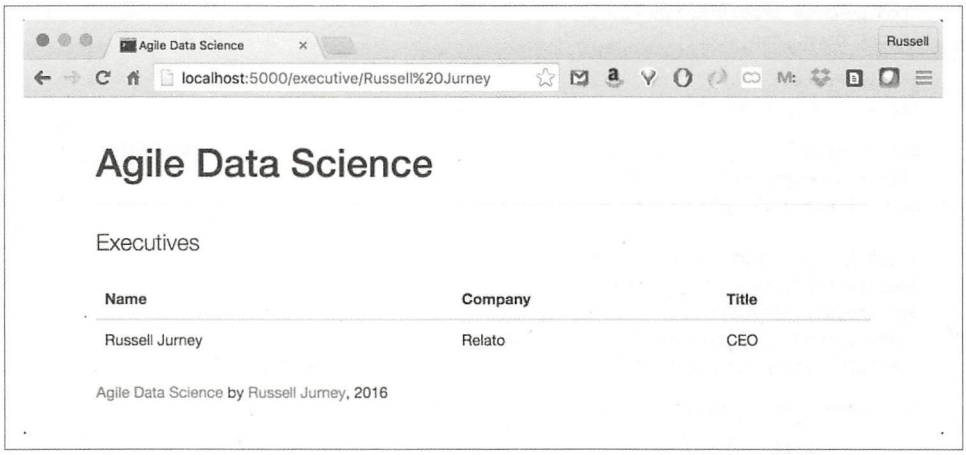


图 2-21　Bootstrap 风格的简易数据表

使用 D3.js 实现数据可视化

D3.js（*https://d3js.org/*）支持数据驱动的文档。它的原作者 Mike Bostock 如是说：

仅供非商业用途或交流学习使用

> d3 并不是传统意义上的可视化框架。d3 没有提供一个包含全部可能需要功能的巨大系统，而是专注于解决问题的难点：基于数据对文档进行高效操控。这使得 d3 有了非比寻常的灵活性，能充分使用底层的 CSS3、HTML5、SVG 等技术的所有能力。

我们将使用 D3.js 创建应用中的图表。和 Bootstrap 一样，D3.js 已经安装在 /static 目录中了。今后我们就要用 D3.js 来画图了。现在，先通过官方的示例合集（*https://github.com/d3/d3/wiki/Gallery*）来看一看 D3.js 能画出怎样的图吧。

本章小结

我们已经遍览了开发环境，并学习了每个工具的入门操作。这些工具共同形成了一条基于分布式系统的数据流水线，能够收集、处理、发布和修饰任意规模的数据。这条流水线的每一个阶段都可以用一行代码轻松修改。这条流水线伸展规模时无须担忧每一步的优化——优化会是一个问题，但不是我们关心的主要问题。

在下一章中我们会看到，由于我们已经创建了一条可以任意伸缩的流水线，其中每个阶段都可以轻松修改，所以我们可能可以恢复敏捷了。我们不会因为从传统数据库切换到一些具有更好伸缩性的平台而遇到绊脚石，也不会因为使用为类似在线事务处理等特定任务专门设计的工具而受到限制。

我们现在有充分的自由去使用这个框架内最好的工具来解决问题，创造价值。我们可以使用任意编程语言、任意编程框架以及任意库，并把它们整合到一起把事情做成。

仅供非商业用途或交流学习使用

第 3 章

数据

本章介绍本书余下部分要使用的数据集。本章也会介绍我们选用的工具的类型，以及我们这样选择的原因。最后，本章会列出我们在分析数据时要用到的一些观点，希望能引发你的进一步思考。

飞行航班数据

乘飞机出行是现代生活中不可或缺的一部分，也是全球化浪潮的基本内容之一，将全球主要城市相互连接起来，形成环球城市经济体。根据一些管理制度的要求，许多航班数据可以供我们免费获取。在本书中，我们会使用许多航班数据集，其中最核心的数据是各次航班起降的准点记录。使用时，我们会结合航线信息、天气、飞行路线等数据一起进行分析。

航班准点记录并不是那么“大”的数据，不过每年增加的未压缩数据也有几个 GB 的量。我们马上就要面对这样一个“大”数据问题（实际上是一个“中”数据问题）——在自己的电脑上直接处理这些数据几乎行不通。要处理这种比电脑内存容量更大的数据集，我们需要用一些具有可伸缩性的工具，这些工具也可以帮助我们学习大数据。我们大家都很熟悉飞机出行，因此我们要使用这些数据让你了解如何分析与查询航班数据，并且了解哪些技术是行之有效的。这可以培养你的*数据直觉*（*data intuition*），也是敏捷数据科学的主题之一。

在本书中，我们所使用的工具都可以处理 PB 级的数据量，不过我们只需要用到单机的本地模式，完全可以在自己的电脑上运行。这些工具不仅让我们可以高效处理数据，还让我们只编译应用一次，应用就可以在各种规模的集群上运行。这简化了我们的各项工作，而简单恰恰是敏捷开发的核心。

仅供非商业用途或交流学习使用

航班准点情况数据

90% ~ 95% 的出发地在美国的航班记录都可以在交通统计局（*https://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236&DB_Short_Name=On-Time*）查到。你可以按月下载这些数据，而我们已经把 2015 年的数据下载好了，你可以在该压缩的 CSV 文件中找到。

数据的字段很多：

```
"Year","Quarter","Month",
"DayofMonth","DayOfWeek","FlightDate","UniqueCarrier",
"AirlineID","Carrier","TailNum","FlightNum",
"OriginAirportID","OriginAirportSeqID","OriginCityMarketID",
"Origin","OriginCityName","OriginState","OriginStateFips",
"OriginStateName","OriginWac","DestAirportID","DestAirportSeqID",
"DestCityMarketID","Dest","DestCityName","DestState",
"DestStateFips","DestStateName","DestWac","CRSDepTime","DepTime",
"DepDelay","DepDelayMinutes","DepDel15","DepartureDelayGroups",
"DepTimeBlk","TaxiOut","WheelsOff","WheelsOn","TaxiIn",
"CRSArrTime","ArrTime","ArrDelay","ArrDelayMinutes",
"ArrDel15","ArrivalDelayGroups","ArrTimeBlk","Cancelled",
"CancellationCode","Diverted","CRSElapsedTime",
"ActualElapsedTime","AirTime","Flights","Distance",
"DistanceGroup","CarrierDelay","WeatherDelay","NASDelay","Security
    Delay","LateAircraftDelay","FirstDepTime","TotalAddGTime",
"LongestAddGTime","DivAirportLandings","DivReachedDest",
"DivActualElapsedTime","DivArrDelay","DivDistance","Div1Airport",
"Div1AirportID","Div1AirportSeqID","Div1WheelsOn",
"Div1TotalGTime","Div1LongestGTime","Div1WheelsOff",
"Div1TailNum","Div2Airport","Div2AirportID",
"Div2AirportSeqID","Div2WheelsOn","Div2TotalGTime",
"Div2LongestGTime","Div2WheelsOff","Div2TailNum","Div3Airport",
"Div3AirportID","Div3AirportSeqID","Div3WheelsOn",
"Div3TotalGTime","Div3LongestGTime","Div3WheelsOff","Div3TailNum",
"Div4Airport","Div4AirportID","Div4AirportSeqID",
"Div4WheelsOn","Div4TotalGTime","Div4LongestGTime",
"Div4WheelsOff","Div4TailNum","Div5Airport","Div5AirportID",
"Div5AirportSeqID","Div5WheelsOn","Div5TotalGTime",
"Div5LongestGTime","Div5WheelsOff","Div5TailNum"
```

下面列出裁剪掉一些列的记录（为了能在本书中更好地显示）：

```
2015,1,1,1,4,2015-01-01,"AA",19805,"AA","N787AA","1",12478,1247802,...,"JFK", ...
2015,1,1,2,5,2015-01-02,"AA",19805,"AA","N795AA","1",12478,...,31703,"JFK", ...
2015,1,1,3,6,2015-01-03,"AA",19805,"AA","N788AA","1",12478,...,31703,"JFK", ...
```

交通统计局网站（BTS）上提供了一份关于表中各字段含义的描述。图 3-1 中展示了其中的一段。在本书中，我们会使用这份描述，作为理解这一大堆字段的参考。

仅供非商业用途或交流学习使用

RITA | BTS | Transtats

www.transtats.bts.gov/Fields.asp?table_id=...

TranStats

Search this site: Go

Advanced Search

Resources

- Database Directory
- Glossary
- Upcoming Releases
- Data Release History

Data Tools

- Table Profile
- Download

: On-Time Performance

Database Profile Data Tables Table Profile

Latest Available Data: January 2016 <<Prev Rows : 1 - 100 of 111 Next>>

Field Name	Description	
Summaries		
*OntimeArrivalPct	Percent of flights that arrive on time. For percent of on time arrivals at specific airports, click Analysis. **Note**: If you select Origin as a category, you get percent of flights that depart from those airports and arrive on time.	Analysis
*OntimeDeparturePct	Percent of flights that depart on time. For percent of on time departures at specific airports, click Analysis. **Note**: If you select Dest as a category, you get percent of flights that depart on time and arrive at those airports.	Analysis
Time Period		
Year	Year	
Quarter	Quarter (1-4)	Analysis
Month	Month	Analysis
DayofMonth	Day of Month	
DayOfWeek	Day of Week	Analysis
FlightDate	Flight Date (yyyymmdd)	
Airline		
UniqueCarrier	Unique Carrier Code. When the same code has been used by multiple carriers, a numeric suffix is used for earlier users, for example, PA, PA(1), PA(2). Use this field for analysis across a range of years.	Analysis
AirlineID	An identification number assigned by US DOT to identify a unique airline (carrier). A unique airline (carrier) is defined as one holding and reporting under the same DOT certificate regardless of its Code, Name, or holding company/corporation.	Analysis
Carrier	Code assigned by IATA and commonly used to identify a carrier. As the same code may have been assigned to different carriers over time, the code is not always unique. For analysis, use the Unique Carrier Code.	
TailNum	Tail Number	
FlightNum	Flight Number	

图 3-1　交通统计局网站上对准点情况数据集中各字段含义的描述

这是一个完全没有被处理过的表，与被正则化处理过的数据相比，这样的数据比较低效。因此，我们更希望得到被正则化处理过的数据。这种未经处理的数据被称为*半结构化数据*（*semistructured data*）。

OpenFlights 数据库

OpenFlights.org 发布了一个关于机场、航空公司、航线相关信息的数据库（*https://openflights.org/data.html*）。我们要在分析中使用这个数据库评估机场的特点。这些数据库中的数据是发布者花钱收集来的，但是发布者还是允许用户免费下载数据库（如果你要把数据用在实际应用中，真心建议你捐款给发布者，毕竟不断收集、更新这样一份无价的数据需要金钱的支持）。

查阅 *download.sh*，我们可以用这个脚本来下载 OpenFlights 数据库，具体如下：

仅供非商业用途或交流学习使用

```
# 获取 openflights 数据
wget -P /tmp/ \
 https://raw.githubusercontent.com/jpatokal/openflights/ \
   master/data/airports.dat
mv /tmp/airports.dat data/airports.csv

wget -P /tmp/ \
 https://raw.githubusercontent.com/jpatokal/openflights/ \
   master/data/airlines.dat
mv /tmp/airlines.dat data/airlines.csv

wget -P /tmp/ \
 https://raw.githubusercontent.com/jpatokal/openflights/ \
   master/data/routes.dat
mv /tmp/routes.dat data/routes.csv

wget -P /tmp/ \
 https://raw.githubusercontent.com/jpatokal/openflights/ \
   master/data/countries.dat
mv /tmp/countries.dat data/countries.csv
```

天气数据

从美国国家环境信息中心（NCEI，前身为美国国家气候数据中心，NCDC，*https://www.ncdc.noaa.gov/*）的网站上可以下载到非常丰富的天气数据，我们用其做分析很方便。

参照脚本 *download_weather.sh*（*http://bit.ly/2nRYGjB*），我们可以用它来下载 WBAN 主站点[1]列表。我们使用列表中的经纬度地理坐标，将机场和气象站点对应起来，用来加强对航班延误的预测。要执行完这个脚本可能比较费时间，所以我们不妨让脚本在后台慢慢执行，稍候再继续工作：

```
cd data

# 将主站点列表作为管道分隔值
curl -Lko /tmp/wbanmasterlist.psv.zip \
  http://www.ncdc.noaa.gov/homr/file/wbanmasterlist.psv.zip
unzip -o /tmp/wbanmasterlist.psv.zip
```

我们还要下载 2015 年所有 WBAN 气象站质量保障的天气观测结果，包括每小时与每天的数据。

```
# 每月获取所有站点的每日摘要文件
# curl -Lko /tmp/ \
  http://www.ncdc.noaa.gov/orders/qclcd/ \
   QCLCD201501.zip
for i in $(seq -w 1 12)
do
  curl -Lko /tmp/QCLCD2015${i}.zip http://www.ncdc.noaa.gov/orders/qclcd/ \
    QCLCD2015${i}.zip
```

1 WBAN 表示 Weather-Bureau-Army-Navy，是一种标注气象台的体系，由于最初的参与者包括美国气象局、美国陆军、美国海军等而得名。——译者注

仅供非商业用途或交流学习使用

```
    unzip -o /tmp/QCLCD2015${i}.zip
done
```

敏捷数据科学中的数据处理

敏捷数据科学中的数据处理同时使用 SQL 查询与 *NoSQL* 数据流编程操作半结构化数据。表结构在运行时动态判定，数据使用 JSON 格式进行序列化处理。这些处理方式使得我们可以不断地将数据提炼为新的结构，从而用于各种需求。

结构化数据 vs. 半结构化数据

维基百科上半结构化数据的定义：

> 结构化数据的一种形式，不同于关系型数据库中有固定结构的数据模型，也不同于其他形式的数据表，然而包含标签或是其他的标记，可以划分数据中的语义元素，明确记录和字段的层级。

这和关系型的结构化数据是截然不同的。在关系型的结构化数据中，数据使用严格的外部表结构描述，拆分为很多表并相互引用避免数据重复。这些工作都是为了后面能进行高效的查询而在分析进行前完成的。用于在线事务处理（Online Transaction Processing，OLTP）的关系型数据库使用高度正则化的表结构，简化了实际业务规则对于数据的定义。

从 20 世纪 70 年代到 21 世纪初，关系型数据库都是数据处理和存储的首选方式。SQL 也成为人们操作结构化数据的首选方式。在 Hadoop 领衔的 NoSQL 运动开始之前，数据处理完全是关系型数据库的天下，而关系型数据库也成为数据处理发展的桎梏。研究机构以外的数据处理完全被关系型系统限死。关系型系统带来的失望与愤怒，构成了 NoSQL 中的“No”。

Hadoop 开发的初衷是解决处理传统关系型数据库无法处理的海量数据的问题，它的数据处理模型也把我们从关系型的表结构中解放了出来。更重要的是，Hadoop 把研究机构的统计推断与研究的工具同行业数据与处理整合到了一起。这样，大数据的潮流使得分析型应用的出现成为可能。

同时，其他一些用于 OLTP 处理的 NoSQL 系统也在许多常见应用中取代了关系型数据库。我们在本书中用来发布数据（与处理数据相对应）的 MongoDB（*https://github.com/mongodb/mongo*）已经成为网络应用的常见选择。

仅供非商业用途或交流学习使用

同样由 O'Reilly 出版的，Seyed M.M. Tahaghoghi 和 Hugh E. Williams 所著的 *Learning MySQL*（*http://oreil.ly/2pq1o0F*）一书中，所使用的航班信息数据库展示了一个结构化、关系型的航班数据视图，见图 3-2。

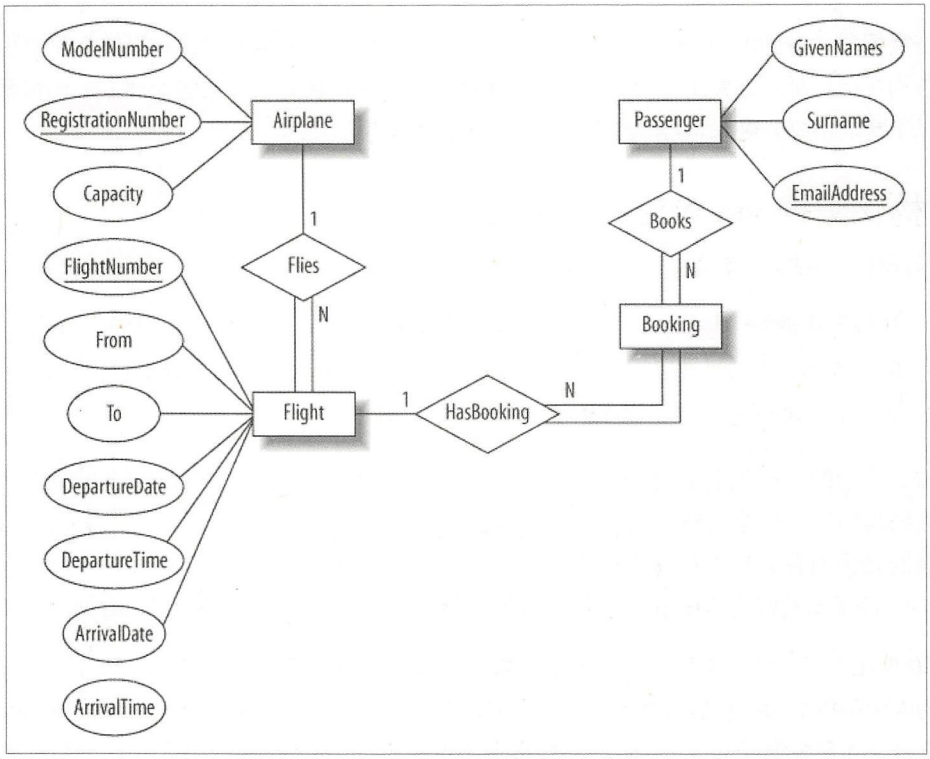


图 3-2　*Learning MySQL* 一书中展示的完全结构化的航班信息数据库

SQL vs. NoSQL

NoSQL 运动把我们从 SQL 的束缚中解放了出来。这是什么意思呢？这意味着 NoSQL 给了我们除关系型数据库中 SQL 以外的数据处理方式。SQL 模式的问题不是 SQL 语言本身，而是单一的数据处理方式，让人们似乎不得不把所有的数据处理交给 SQL，而无视了数据本身是否真的适合这种方式。

在本书第 1 版中，我们仿照 NoSQL 社区中许多书的做法，除了讲述 SQL 不适合构建数据分析应用这件事，还在其他部分完全回避了 SQL。在这一版中，我们决定讲得更广一些，尤其随着像 Spark 这样的新工具出现，SQL 和 NoSQL 已经逐渐统一。人们开始意识到只要有足够多的数据处理方式可选，SQL 本身其实也是非常有用的。SQL 和 NoSQL 在敏捷数据科学中都很重要，让我们走着瞧吧。

SQL

SQL 在构建数据分析应用时到底会发挥什么作用呢？要查询关系型、结构化的数据，我们通常使用 SQL 这类声明式的编程语言。在 SQL 中，我们只要声明我们想要的是什么，而不是写要怎么获得我们想要的。这和 Java、Scala、Python 等语言所属的命令式的编程迥然不同。用 SQL 我们只要写我们要的输出，而不是对数据的一组具体操作。

当进行快速简单的即席查询时，SQL 是非常高效的。比如，我们要在图 3-2 中的表上查询 2015 年 1 月 1 日按起降城市统计的航班计数的话，我们只需写这样的语句：

```
SELECT From, To, COUNT(*)
    FROM Flight
      WHERE DepartureDate == '2015-01-01'
        GROUP BY From, To;
```

这种声明式的编程非常适用于提炼和查询结构化数据，聚合数据生成简单的图表。当我们知道我们想要什么时，我们可以高效地告诉 SQL 引擎，SQL 引擎就可以为我们求出结果表。查询的具体执行过程不需要我们操心。

SQL 有两大不足之处。首先，我们要依赖数据库来决定如何执行查询操作，这对于查询来说可能是好事也可能是坏事，取决于具体的查询操作。对于大数据场景，这就有问题了。如果查询计划器不好，我们可能会永远也等不到查询完成的时候。也就是说，对于海量数据，有时我们必须参与到最佳查询计划的决策中，不能依赖传统的查询计划器。使用 PySpark，只要我们想做，就可以通过数据流编程指定各操作具体该怎么完成。在不想这么做的时候，PySpark 的 SQL 抽象也可以为我们生成执行计划。我们可以兼得两种方式的优点。

其次是复杂性。一旦查询过于复杂，SQL 语句就非常难以看懂。查询语句嵌套的子查询中还嵌套着子查询，这意味着代码不够直观。对于复杂操作，命令式的代码比声明式的更加易读、易理解。尽管在关系型系统中我们也可以把复杂查询划分为多个阶段，但这并不是经过优化的做法。

当 SQL 是唯一采用的方式时，这些不足之处会让人们觉得它很不好用。不过，既然我们已经有了其他的数据处理方式，那么 SQL 又可以重新成为大家的好帮手。对于简单的查询来说，SQL 强大、简单、易学，优点明显。

NoSQL 与数据流编程

与 SQL 相反，在构建分析型应用时，我们常常并不知道要执行的查询是什么，因此我们无法写出查询语句，需要许多次试验和迭代才能得到问题的解。数据也常常不是以关系型的

形式存在的。很多数据尚未处理，仍是杂乱无章的脏数据。提取这些数据的结构是一个冗长的过程，因此我们选择在迭代中依次提取不同的特征。直接获取数据的结构是不现实的。

出于这样的原因，在敏捷数据科学中，我们经常要使用运行于分布式系统上的命令式的语言。Python 和 PySpark 这种命令式的工具让我们可以描述操作数据的步骤。我们要用到并行的多处理核来暴力读取数据记录以快速执行，而不是使用基于我们还没获得的结构才能预计算的索引。Spark（以及之前的 Hadoop）使这一切成为可能。

除了可以很好地利用 Hadoop 和 Spark 等技术让我们轻松扩展数据处理，命令式语言还让我们把重点放在构建分析型应用的主要任务上：以迭代和增量的方式搞定艰难而关键的步骤。这也是让我们的应用能够发掘价值的关键所在，是一个内在的命令式过程。

和写 SQL 语句相比，实现这些精巧的命令式操作是一个漫长而曲折的过程，我们要用到包括统计学、机器学习、社会科学等在内的技术。这种任务适合命令式编程。

Spark: SQL + NoSQL

所以，SQL 是为查询数据而优化设计的，而针对数据流的工具则是为提炼数据而优化设计的。我们既需要查询数据（对数据提问），也需要处理数据（从一个或多个数据源中计算出一些新东西）。幸运的是，Spark 同时支持这两种编程范式！这是 Spark 接口中最大的创新。这个特性使得我们可以在声明式的 SQL 语句和命令式的 Python 语句中随心所欲地自由切换。这是 Spark 的一大好处，相对于 Hadoop 是一个巨大的进步。之前，Pig（Hadoop 软件栈中的数据流编程工具）和 Hive（Hadoop 软件栈中的 SQL）是两个相互独立的工具，而且很不幸的是这两个工具社区之间还有些不友好。

NoSQL 中的表结构

如果表结构一成不变，SQL 又是我们唯一的工具，我们的想法就会被这些为消耗数据而优化的工具所主导，而不是去挖掘数据。为数据表声明严格的表结构，是我们做好事情的阻碍，我们找到数据间内在联系的能力被限制了。而使用半结构化的数据则让我们可以专注于数据本身，迭代地操作数据，提炼出价值并转化为产品。

我们使用数据流编程语言在代码中定义数据的格式，然后使用 SQL 查询数据，或直接发布到文档存储中。这些都不需要正式指定表结构！表结构信息随数据存在，是内在的而不是外在的。这是为数据科学而优化的，我们可以从多个已有的数据源中发掘出新的信息。在这种场景下，在外部声明表结构没有任何好处，反而是一种累赘。毕竟，我们只有在结果出现时才知道会得到什么，数据科学处处有惊喜！

数据序列化

尽管我们可以直接操作纯文本格式的半结构化数据，但是使用一些包含表结构信息的格式还是可以帮助强化原始记录的某些结构化信息。序列化系统可以让我们做到这点。下面列举了一些可以使用的序列化系统：

- Thrift（*http://thrift.apache.org*）
- Protobuf（*http://code.google.com/p/protobuf/*）
- Avro（*http://avro.apache.org/*）

在本书第 1 版中，我们选择了 Avro。Avro 支持复杂的数据结构，Avro 格式的每个文件中都包含一份表结构信息，Avro 还支持许多工具直接读取，可以使用多种编程语言访问。然而，我们在使用一些不同的编程语言访问 Avro 数据时，经常会遇到一些实现上的错误，这严重影响了我们的产出。总是会让我们在一些无意义的问题上花费许多精力。因此，在第 2 版中，我们准备使用 JSON 行格式来代替 Avro. JSON 行格式（JSON Lines）（*http://jsonlines.org/*）。JSON 行格式也叫作换行符分隔的 JSON 格式（newline- delimited JSON（NDJSON））（*http://ndjson.org/*），这种格式很简单，一条 JSON 记录会被存储为文本中的一行。

动态结构表的特征提取与呈现

正如 Pete Warden 在他的讲座“拥抱混沌的数据”（Embracing the Chaos of Data）中提到的，大多数可以免费得到的数据都很粗糙而且是非结构化的。这些难看的、没有仔细清理并正则化的数据表有很多，也很容易获得，正是对这些数据的处理，才称得上“大数据”。在这些大数据中蕴含着许多机遇，比如，从原始数据中挖掘出提炼过的信息，使用这些提炼过的信息来为一些行为提供新的决策思路。

从非结构化数据中提取出的特征只有在“强光曝晒”下才会变得更好，因为用户使用了这些特征，如果不好用会进行投诉；如果在提取特征后无法发布出来，那么就像是处于自由的状态。构建数据产品中最难的部分就是把提取出的实体和特征限定到比想象的小得多的产品中。这就是为什么表结构在一开始要使用非结构化的文本，经过特征提取后才演进为结构化数据。

特征必须在生成时就以某种产品的形式呈现，否则将永远无法达到可以用于实际决策的状态。产品内部的中间数据不会自行改善。最好创建出实体页面，让数据逐渐达到“客户级别”

的形式，不断改进这些实体，逐步将它们组合起来，而不是试图从一开始就以宏大的视角展示数不清的中间数据。

在将数据编排成结构良好的信息的同时，利用这些信息揭示新的结论并基于它们做出可以影响决策的预测，这一过程蕴含非凡潜力，可以创造巨大的价值。数据是残酷无情的，假如不能发掘出数据的本质的话，再雄心勃勃的产品经理也只能“望数据兴叹”。

我们会在整本书中看到，表结构在不断演进和优化，而发掘出表结构的特征也在不断演进。它们的并行演进，也是我们敏捷数据科学的敏捷之处。

本章小结

我们在本书中需要使用的主要数据集就这么介绍完了。随着工作的推进，我们还会引入一些其他的数据集。在下一章中，我们就要开始攀登数据价值金字塔了！

第 II 部分

攀登金字塔

如果你面前的路每一步都一清二楚，你就知道这不是属于你自己的路。真正属于自己的路是自己一步一步开拓出来的。唯有这样的路才配称为自己的路。

——约瑟夫·坎贝尔

第 II 部分介绍的是本书剩下的内容——数据价值金字塔（data-value pyramid）的结构。在我们接下来的课程中，我们会从简单的记录开始，最终得到交互式的预测模型，在不断迭代的过程中攀登数据价值金字塔。我们从理论开始讲，然后使用之前介绍的这套框架，深入数据科学中去实践。

构建敏捷数据科学的产品就是构建一个场景，在这个场景中可以导出可重现的推论，并且被不断巩固，最终根据这些推论发掘出数据的价值。其中最初级的内容就是记录的展示。而最终的内容则是要从数据中创造并获取价值。这一过程是一个充满发现的过程。

这一过程的结构就是我们所说的数据价值金字塔，见图 II-1。

数据价值栈借鉴了马斯洛需求层次理论中，下层必须为上层的前置条件的概念。上层内容（比如预测模型）依赖于下层内容（比如数据报表），所以我们不能跳过前面的内容。如果我们直接处理上层，就会对数据及其结构缺乏足够的理解，无法轻易构建出上层所需的特性与数值。

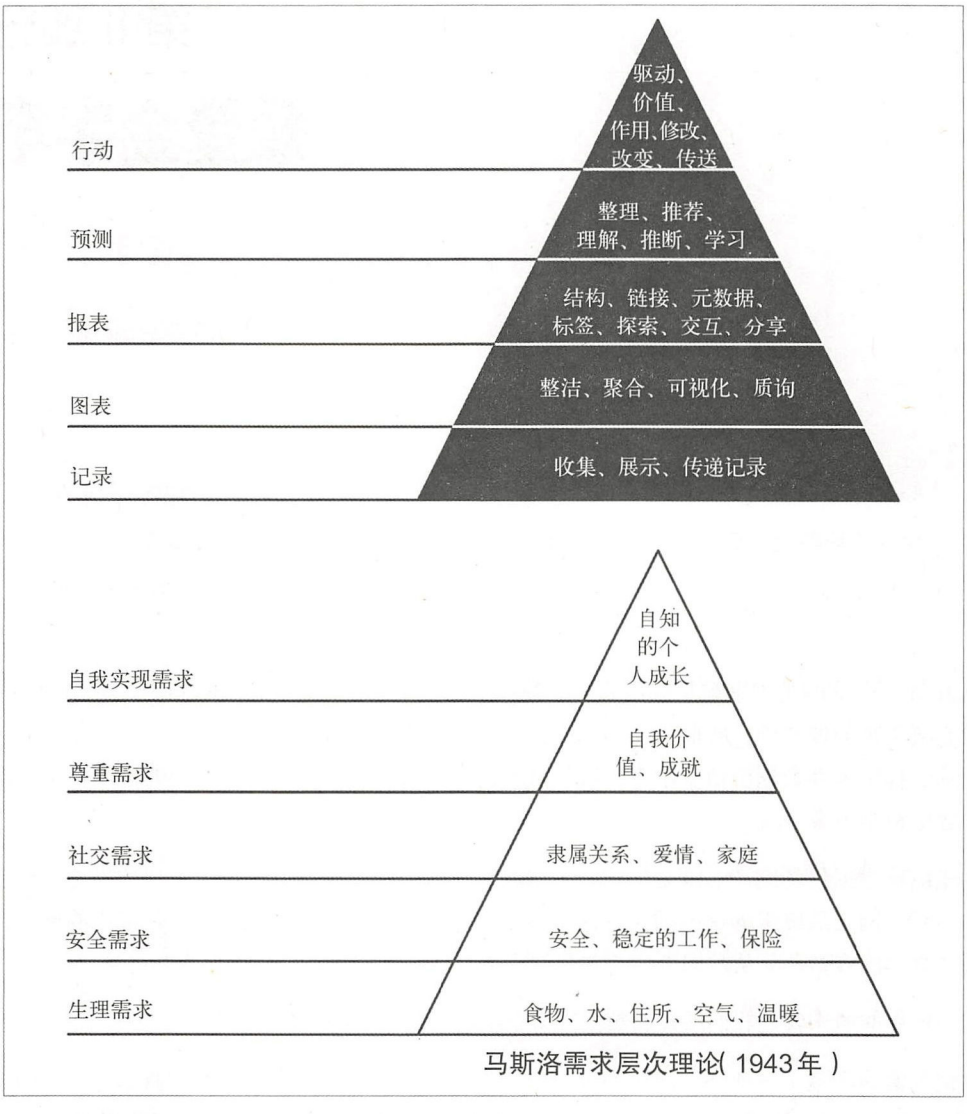


图 II-1　Jurney—Warden 数据价值金字塔 (2011 年)

数据价值栈的底层是数据记录的简单展示，这一环节的关键是为数据构建出一条流水线，将原始数据转为用户屏幕上呈现走通的流程。接下来就是图表部分，我们对数据进行处理，获取结构化的数据并对数据的一些维度进行聚合，熟悉数据的各项属性。下一步就是识别数据间的关联，通过交互式报表探索数据。这样我们就可以使用基于统计的推断来生成预测模型。最终，预测模型给出预测，用户因此做出相应的行为改变，这样整个过程就创造并取得了价值。

第 4 章

记录收集与展示

本章是我们敏捷开发周期的第一个冲刺（sprint），我们要攀登数据价值金字塔的第一层（见图 4-1）。我们要把从原始数据一步一步转为最终用户能看的网络应用的过程走通。这样单个开发者也可以把原始数据发布到页面上了。通过这一步，我们就在真实数据上试用了整个软件栈，将我们的应用程序一头连上数据本质，一头连上用户。

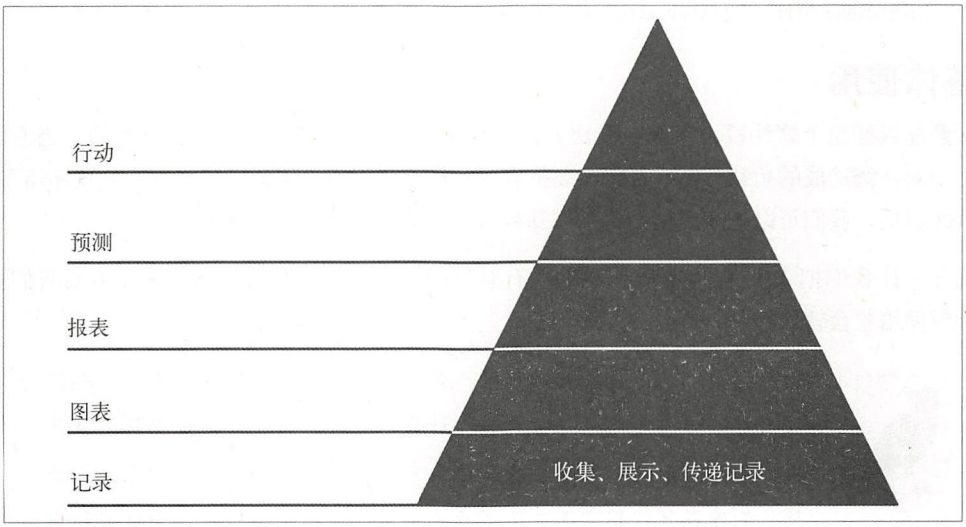


图 4-1　第一层：展示基础记录

如果你开发过广受欢迎的应用程序，这一步骤可能会让你觉得多余，毕竟你已经会在应用中展示单条记录（原子记录）。那么对你来说，这一步骤的意义就在于走通从大容量存储中的数据一直到浏览器中的内容之间一步步的分析转化过程。通过数据抽取、转化、加载

（ETL）或其他某些方式，大容量存储可以支持进一步的数据处理。

数据价值金字塔的第一层可以过得比较快，以便进一步探索金字塔上层的价值。需要注意的是，由于还需要收集更多的数据集来使我们的分析更加丰富，所以我们以后依然会不断涉及这一层的内容。随着不断深入，我们需要探索更多的数据集。在本书中，我们不断向金字塔上层前进，整个过程都需要数据的收集与展示。我们在分析与听取用户反馈的同时，需要不断用到数据价值金字塔的各个层次。在我们进一步探索数据科学金字塔时，复杂度与获得的价值都如滚雪球般不断增大，这样的配置以及实时可浏览的记录为这一切提供了基础。

如果你有 PB 级的原子记录，把这些记录全放到文档存储中可能并不方便。更何况数据安全方面的限制也可能禁止导出全部数据。因此，我们需要拿一些采样数据。准备一份较小的采样数据并发布出来，之后就只使用这份采样数据。

本章的代码示例可以在 *Agile_Data_Code_2/ch04* 目录中找到。克隆（Clone）代码仓库，跟上节奏！

```
git clone https://github.com/rjurney/Agile_Data_Code_2.git
```

整体使用

安装配置好整个软件栈要花费一番功夫。这番功夫不是白费的，有了这套软件栈，当看到用户对系统造成的负载升高，系统需要扩容的时候，我们就不用慌慌张张重新配置环境了。从此以后，我们可以把全部精力都放在迭代开发上，不断提升产品本身。

现在，让我们把 2015 年从美国起飞的所有航班的起降时间信息数据拿出来，看看我们要如何使用整套软件栈。

一条原子记录是一条基础记录，是我们分析事件时的最小粒度。我们可以对一组原子记录进行聚合、计数、分段、分块，但是单条记录是不可分为多条的。因此，原子记录表示的是基本事实，操作原子记录是我们理解数据的实际意义并构建应用程序的关键所在。大数据的核心就是能够使用 NoSQL 工具，对这些最琐碎的数据进行分析，并获得原先不可能达到的深层次理解。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

航班数据收集与序列化

在图 4-2 中，我们可以看到事件序列化的过程。因此，我们要把本书接下来会用到的核心数据下载下来，参见 *download.sh*：

```
# 获取 2015 年所有航班的准点数据—— 273MB
wget -P data/ \
 http://s3.amazonaws.com/agile_data_science/ \
   On_Time_On_Time_Performance_2015.csv.bz2

# 从 openflights 中获取数据
wget -P /tmp/\
 https://raw.githubusercontent.com/jpatokal/openflights/ \
  master/data/airports.dat
mv /tmp/airports.dat data/airports.csv

wget -P /tmp/\
 https://raw.githubusercontent.com/jpatokal/openflights/\
  master/data/airlines.dat
mv /tmp/airlines.dat data/airlines.csv

wget -P /tmp/ \
 https://raw.githubusercontent.com/jpatokal/openflights/\
  master/data/routes.dat
mv /tmp/routes.dat data/routes.csv

wget -P /tmp/ \
 https://raw.githubusercontent.com/jpatokal/openflights/\
  master/data/countries.dat
mv /tmp/countries.dat data/countries.csv

# 获取 FAA 数据
wget -P data/ http://av-info.faa.gov/data/ACRef/tab/aircraft.txt
wget -P data/ http://av-info.faa.gov/data/ACRef/tab/ata.txt
wget -P data/ http://av-info.faa.gov/data/ACRef/tab/compt.txt
wget -P data/ http://av-info.faa.gov/data/ACRef/tab/engine.txt
wget -P data/ http://av-info.faa.gov/data/ACRef/tab/prop.txt
```

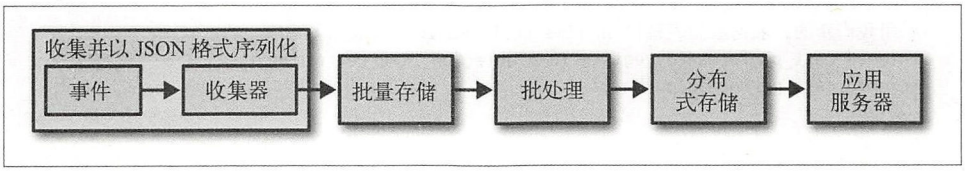


图 4-2　事件序列化

作为开始，我们先把航班准点数据记录中不需要的列裁剪掉，然后把得到的数据转换为 Parquet 格式。这样可以提高装载数据时的性能，毕竟我们接下来需要经常装载数据。而在实际中，你可能要保留任何将来可能会有用的数据。注意，`spark-csv` 的 `inferSchema` 参

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

数有错误，所以我们要在保存处理过的数据之前，手动转换数值型字段的数据类型。

如果你想看懂下面这条查询，请参阅交通统计局对准点情况记录（这份数据在第 3 章中介绍过）的数据描述（*https://www.transtats.bts.gov/Fields.asp?Table_ID=236*）。

运行下面的代码，可以将数据中我们用不到的列都裁剪掉：

```
# 读取 CSV 文件，并设置打开头解析与类型推断，只要一条语句！
on_time_dataframe = spark.read.format('com.databricks.spark.csv')\
   .options(
     header='true',
     treatEmptyValuesAsNulls='true',
   )\
   .load('data/On_Time_On_Time_Performance_2015.csv.bz2')
on_time_dataframe.registerTempTable("on_time_performance")

trimmed_cast_performance = spark.sql("""
SELECT
  Year, Quarter, Month, DayofMonth, DayOfWeek, FlightDate,
  Carrier, TailNum, FlightNum,
  Origin, OriginCityName, OriginState,
  Dest, DestCityName, DestState,
  DepTime, cast(DepDelay as float), cast(DepDelayMinutes as int),
  cast(TaxiOut as float), cast(TaxiIn as float),
  WheelsOff, WheelsOn,
  ArrTime, cast(ArrDelay as float), cast(ArrDelayMinutes as float),
  cast(Cancelled as int), cast(Diverted as int),
  cast(ActualElapsedTime as float), cast(AirTime as float),
  cast(Flights as int), cast(Distance as float),
  cast(CarrierDelay as float), cast(WeatherDelay as float),
  cast(NASDelay as float),
  cast(SecurityDelay as float),
  cast(LateAircraftDelay as float),
  CRSDepTime, CRSArrTime
FROM
  on_time_performance
""")

# 用我们新的、裁剪后的表替换 on_time_performance 表并展示其内容
trimmed_cast_performance.registerTempTable("on_time_performance")
trimmed_cast_performance.show()
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

这样就出现了一个更简化的格式（此处仅展示局部）：

Year	Quarte	Month	DayofMont	DayOfWeek	FlightDate	Carrier	TailNum	FlightNu	Origin
2015	1	1	1	4	2015-01-01	AA	N001AA	1519	DFW
2015	1	1	1	4	2015-01-01	AA	N001AA	1519	MEM
2015	1	1	1	4	2015-01-01	AA	N002AA	2349	ORD
2015	1	1	1	4	2015-01-01	AA	N003AA	1298	DFW
2015	1	1	1	4	2015-01-01	AA	N003AA	1422	DFW

让我们确保数值类型的字段已经转为我们需要的数据类型：

```
# 验证我们能对数值列进行求和操作
spark.sql("""SELECT
  SUM(WeatherDelay), SUM(CarrierDelay), SUM(NASDelay),
  SUM(SecurityDelay), SUM(LateAircraftDelay)
FROM on_time_performance
""").show()
```

这条查询的输出如下（已经根据页面调整过格式）：

sum(WeatherDelay)	sum(CarrierDelay)	sum(NASDelay)	sum(SecurityDelay	um(LateAircraftDelay)
3100233.0	2.0172956E7	1.4335762E7	80985.0	2.4961931E7

在完成数据裁剪与字段类型转换并确保数值列可以正常使用之后，我们就可以用 JSON 和 Parquet 格式存储数据了。注意，我们又把数据读取回来，以验证是否能正确读取。确保这段代码运行时没有报错，因为整本书剩下的内容中都会用到这些文件：

```
# 用 gzip 压缩的 JSON 格式存储记录
trimmed_cast_performance.toJSON()\
  .saveAsTextFile(
    'data/on_time_performance.jsonl.gz',
    'org.apache.hadoop.io.compress.GzipCodec'
)

# 浏览文件系统中的记录
gunzip -c data/On_Time_On_Time_Performance_2015.jsonl.gz/part-00000.gz | head

# 使用 Parquet 格式存储数据
trimmed_cast_performance.write.parquet("data/on_time_performance.parquet")

# 把 JSON 记录读取回来
on_time_dataframe = spark.read.json('data/on_time_performance.jsonl.gz')
on_time_dataframe.show()

# 把 Parquet 文件读取回来
on_time_dataframe = spark.read.parquet('data/trimmed_cast_performance.parquet')
on_time_dataframe.show()
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

你可能留意到 Parquet 文件的大小只有 248 MB，而 gzip 压缩的 CSV 文件有 315 MB，gzip 压缩的 JSON 文件有 259 MB。事实上，使用 Parquet 格式的性能要好得多，因为它只会读取那些我们在 PySpark 脚本中实际用到的数据列。

我们可以用 `gunzip-c` 和 `head` 命令查看 gzip 压缩的 JSON 文件：

```
gunzip -c data/On_Time_On_Time_Performance_2015.jsonl.gz/part-00000.gz | head
```

现在我们可以直接查看准点情况记录了，这样我们可以更容易地理解这些数据：

```
{
  "Year":2015,
  "Quarter":1,
  "Month":1,
  "DayofMonth":1,
  "DayOfWeek":4,
  "FlightDate":"2015-01-01",
  "UniqueCarrier":"AA",
  "AirlineID":19805,
  "Carrier":"AA",
  "TailNum":"N787AA",
  "FlightNum":1,
  ...
}
```

使用 PySpark 读取 gzip 压缩的 JSON 格式数据很简单，只需使用 SparkSession 类型的变量 `spark`：

```
# 把 JSON 记录读取回来
on_time_dataframe = spark.read.json('data/On_Time_On_Time_Performance_2015.jsonl.gz')
on_time_dataframe.show()
```

读取 Parquet 数据也一样简单：

```
# 读取 Parquet 文件
on_time_dataframe = spark.read.parquet('data/on_time_performance.parquet')
on_time_dataframe.first()
```

航班记录处理与发布

在收集了航班数据之后，让我们对数据进行处理（见图 4-3）。为了获取初始状态，使用我们的整套软件栈逐步深入探索真实数据，首先让我们把航班准点记录直接发布到 MongoDB 和 Elasticsearch 中，这样我们就可以从网页上直接使用 Mongo、Elasticsearch 以及 Flask 等工具直接访问数据了。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

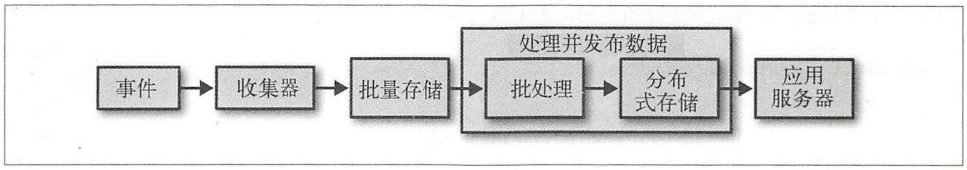


图 4-3　处理与发布数据

把航班记录发布到 MongoDB

MongoDB 的 Spark 支持简化了这一步骤。我们只需要引入并激活 pymongo_spark 包，把 DataFrame 转为 RDD，然后调用 saveToMongoDB 方法。这些代码可以在 *ch04/pyspark_to_mongo.py* 中找到：

```
import pymongo
import pymongo_spark
# 重要：激活 pymongo_spark
pymongo_spark.activate()

on_time_dataframe = spark.read.parquet('data/on_time_performance.parquet')

# 注意，我们要把记录转为 dict
# 以避免 https://jira.mongodb.org/browse/HADOOP-276
as_dict = on_time_dataframe.rdd.map(lambda row: row.asDict())
as_dict.saveToMongoDB
  ('mongodb://localhost:27017/agile_data_science.on_time_performance')
```

如果哪一步出错了，请直接删掉这个数据集，然后重做一遍，删除方法如下：

```
$ mongo agile_data_science
>db.on_time_performance.drop()

true
```

我们的基础架构方案的优点就在于，任何东西都是可以从原始数据中重算的，因此几乎不用担心数据库崩溃（尽管我们没有使用可以容错的集群）。而且，由于我们使用数据库作为文档存储，可以直接通过 ID 或别的什么来提取文档，因此我们也就不需要担心性能问题了。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

最后，让我们确认航班记录已经在 MongoDB 中存好了：

```
>db.on_time_performance.findOne()
{
        "_id" : ObjectId("56f9ed67b0504718f584d03f"),
        "Origin" : "JFK",
        "Quarter" : 1,
        "FlightNum" : 1,
        "Div4TailNum" : "",
        "Div5TailNum" : "",
        "Div2TailNum" : "",
        "Div3TailNum" : "",
        "ArrDel15" : 0,
        "AirTime" : 378,
        "Div5WheelsOff" : "",
        "DepTimeBlk" : "0900-0959",
       ...
}
```

执飞航空公司、航班日期、航班号是航班记录的最小唯一标识符。现在我们就用这些信息提取出一条航班记录：

```
>db.on_time_performance.findOne(
 {Carrier: 'DL', FlightDate: '2015-01-01', FlightNum: 478})
```

你可能发现这条查询的返回速度并不快。Mongo 允许我们使用列的各种组合查询数据，但是这个功能的开销比较大。我们需要为我们的查询设计并维护索引。在这种查询中，我们访问的模式是固定的，所以很好定义我们的索引：

```
>db.on_time_performance.ensureIndex({Carrier: 1, FlightDate: 1, FlightNum: 1})
```

执行这条命令可能需要一段时间，但是执行完之后我们的查询就会很快了。相对于 Mongo 提供的强大功能而言，建索引的开销还是很小的。总的来说，我们用到数据的功能越多，我们操作时的额外代价就越高。因此，尽量少用数据库的各种功能，除非你很喜欢对生产环境中的数据库进行性能调优。

在浏览器中展示航班记录

现在我们已经把航班准点信息的记录发布到文档存储中，而且可以查询数据了，是时候把我们的数据通过一个简易的网络应用在浏览器中展示出来了（见图 4-4）。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

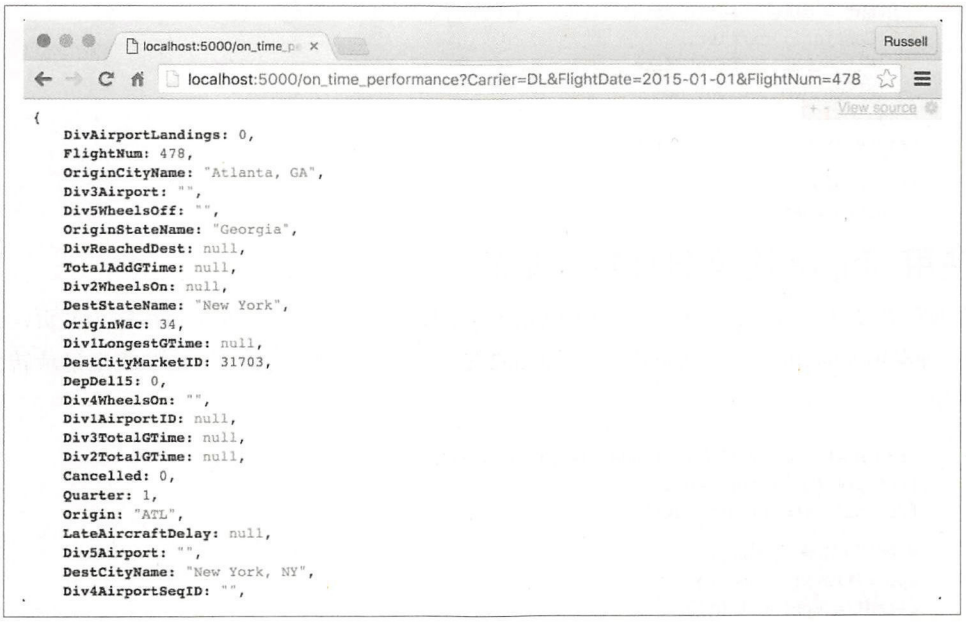


图 4-4　显示一条未经处理的航班记录

使用 Flask 和 pymongo 提供航班信息

Flask 和 pymongo 简化了查询航班信息并获取返回值的过程。*ch04/web/on_time_flask.py* 可以把一条航班记录以 JSON 格式返回到网页上。这段代码可以当作 API 来使用，本书在后面也会创建和使用一系列 JSON 格式的 API。注意，我们不能调用 `json.dumps()`，因为这样涉及以 JSON 格式序列化 pymongo 的记录，而 `json` 并不知道应当如何进行序列化。应该使用的是 `bson.json_util.dumps()`：

```
from flask import Flask, render_template, request
from pymongoimport MongoClient
from bson import json_util

# 配置 Flask 和 Mongo
app = Flask(__name__)
client = MongoClient()

# 控制器：获取一条航班记录并展示
@app.route("/on_time_performance")
def on_time_performance():

  carrier = request.args.get('Carrier')
  flight_date = request.args.get('FlightDate')
  flight_num = request.args.get('FlightNum')
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

```
  flight = client.agile_data_science.on_time_performance.find_one({
   'Carrier': carrier,
   'FlightDate': flight_date,
   'FlightNum': flight_num
  })
return json_util.dumps(flight)

if __name__ == "__main__":
  app.run(debug=True)
```

使用 Jinja2 渲染 HTML5 页面

和我们在第 3 章中做的一样，我们可以用 Jinja2 模板把原始的 JSON 字符串转为网页，详见 *ch04/web/on_time_ ask_template.py*。Jinja2 使得我们可以轻松地把原始的航班记录转为网页：

```
from flask import Flask, render_template, request
from pymongo import MongoClient
from bson import json_util

# 配置 Flask 和 Mongo
app = Flask(__name__)
client = MongoClient()

# 控制器：获取一条航班记录并展示
@app.route("/on_time_performance")
def on_time_performance():

  carrier = request.args.get('Carrier')
  flight_date = request.args.get('FlightDate')
  flight_num = request.args.get('FlightNum')

  flight = client.agile_data_science.on_time_performance.find_one({
    'Carrier': carrier,
    'FlightDate': flight_date,
    'FlightNum': int(flight_num)
  })

  return render_template('flight.html', flight=flight)
if __name__ == "__main__":
  app.run(debug=True)
```

注意，例子中的 `render_template` 定义在文件 *ch04/web/templates/flight.html* 中。这是一个用于向我们的页面布局中的动态内容区域填充内容的局部模板。在它所继承的父模板 *ch04/web/templates/layout.html* 中，引入了 Bootstrap，并且处理了每个页面的全局通用内容设计，比如，页面头部、整体风格、页脚等。这样可以让我们不需要在每个页面间不断重复，就为整个应用创建出风格统一的页面布局。

仅供非商业用途或交流学习使用

布局模板中包含空的内容块 {% block content %}{% end block %}，我们把包含应用数据的局部页面模板填充到这样的内容块中进行渲染：

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Agile Data Science</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta name="description"
            content="Chapter 5 example in Agile Data Science, 2.0">
    <meta name="author" content="Russell Jurney">
    <link href="/static/bootstrap.min.css" rel="stylesheet">
    <link href="/static/bootstrap-theme.min.css" rel="stylesheet">
  </head>

  <body>
    <div id="wrap">
      <!-- Begin page content -->
      <div class="container">
        <div class="page-header">
          <h1>Agile Data Science</h1>
        </div>
        {% block body %}{% endblock %}
      </div>

      <div id="push"></div>
    </div>

    <div id="footer">
      <div class="container">
        <p class="muted credit">
         <a href="http://shop.oreilly.com/product/ \
           0636920025054.do">\
            Agile Data Science</a>by \
             <a href="http://www.linkedin.com/in/ \
              russelljurney">Russell Jurney</a>, 2016
      </div>
    </div>
    <script src="/static/bootstrap.min.js"></script>
  </body>
</html>
```

航班专用的模板可以通过继承基础布局模板实现。Jinja2 模板会执行 {% %} 标签中的内容所定义的控制流，遍历相应的元组与数组，选择对应的条件子句。我们可以在 {{ }} 标签中写绑定的数据或任意的 Python 代码来在页面上显示变量的值。具体如航班模板所示：

```
{% extends "layout.html" %}
{% block body %}
  <div>
    <p class="lead">Flight {{flight.FlightNum}}</p>
    <table class="table">
      <thead>
        <th>Airline</th>
        <th>Origin</th>
        <th>Destination</th>
        <th>Tail Number</th>
        <th>Date</th>
        <th>Air Time</th>
        <th>Distance</th>
      </thead>
      <tbody>
        <tr>
          <td>{{flight.Carrier}}</td>
          <td>{{flight.Origin}}</td>
          <td>{{flight.Dest}}</td>
          <td>{{flight.TailNum}}</td>
          <td>{{flight.FlightDate}}</td>
          <td>{{flight.AirTime}}</td>
          <td>{{flight.Distance}}</td>
        </tr>
      </tbody>
    </table>
  </div>
{% endblock %}
```

主体内容块是用来渲染数据所对应的页面内容的。我们从初始模板开始，对接上数据中的值（通过我们在模板中绑定的 flight 变量），然后就可以在页面上展示记录了。

我们可以看到网页上的航班数据，包括 Carrier、FlightDate 和 FlightNum 等字段。先从 MongoDB 中取出一条航班记录来测试一下：

```
$ mongo agile_data_science
>db.on_time_performance.findOne()

{
  "_id" : ObjectId("56fd7391b05047327f19241f"),
  "Origin" : "IAH",
  "FlightNum" : 1044,
  "Carrier" : "AA",
  "FlightDate" : "2015-07-03",
  "DivActualElapsedTime" : null,
  "AirTime" : 122,
  "Div5WheelsOff" : "",
  "DestCityMarketID" : 32467,
  "Div3AirportID" : null,
  "Div3TotalGTime" : null,
```

```
    "Month" : 7,
    "CRSElapsedTime" : 151,
    "DestStateName" : "Florida",
    "DestAirportID" : 13303,
    "Distance" : 964,
    ...
  }
```

这样我们就可以获取一条航班记录，并展示在 */ch04/web/templates/layout.html* 所定义的页面布局中了，见图 4-5。

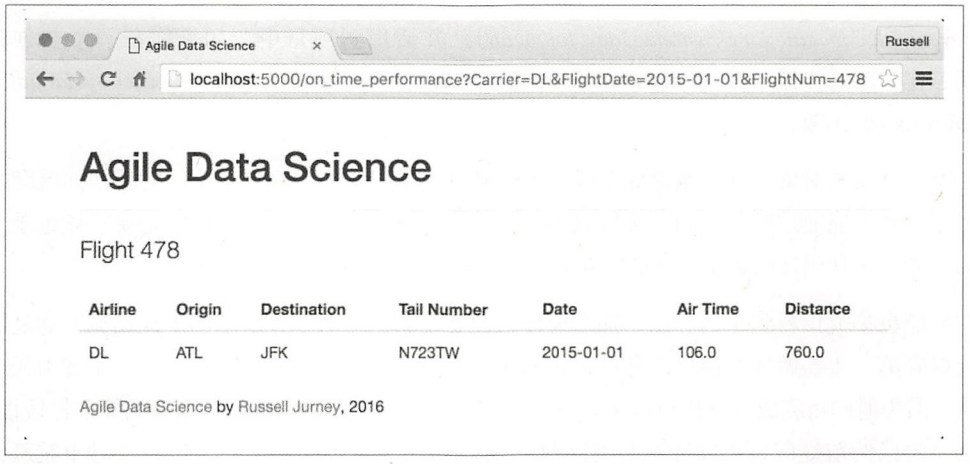


图 4-5　展示单条航班数据

在 Flask 控制台上可以看到被访问的资源（由于页面宽度限制，删掉了日期和时间戳）：

```
127.0.0.1 - - [...]
   "GET /on_time_performance?Carrier=DL& \
     FlightDate=2015-01-01&FlightNum=478 HTTP/1.1" 200 -
127.0.0.1 - - [...] "GET /static/bootstrap.min.css HTTP/1.1" 200 -
127.0.0.1 - - [...] "GET /static/bootstrap-theme.min.css HTTP/1.1" 200 -
127.0.0.1 - - [...] "GET /static/bootstrap.min.js HTTP/1.1" 200 -
127.0.0.1 - - [...] "GET /favicon.ico HTTP/1.1" 404 -
```

很好！我们从原始数据搞出了一个网页！但是……那又怎样呢？我们究竟获得了什么？

我们已经完成了原子记录的展示，这是金字塔最基本的底层，也是实现标准数据流水线的第一步。这是一切的基础。不管我们提供什么样的高级分析功能，归根结底用户还是希望看到支撑我们的结论的数据本身。这一步骤跳不得，如果我们不能正确地可视化一条原子记录，那么我们的平台和策略就失去了根本，因而站不住脚。

敏捷开发检查站

既然我们有了能运行的软件，那么是时候让用户试用并提供反馈了。“等等，真的吗？这也太早了！”别那么想了！

谁都想成为史蒂夫·乔布斯；谁都想开一个酷炫的产品发布会，把绝密的新发明拿出来引起轰动。但是对于分析型应用而言，如果迟疑而不敢发布，脆弱的自我会削弱你成为乔布斯的能力，因为会担心第一个版本没有达到乔布斯的水平。如果不把蹩脚的第一步发布出来，那么优秀的第二十六步就更难实现了。强烈建议你学一学“客户发展”（*https://steveblank.com/category/customer-development/*），并活用到项目中。如果你在一个初创公司工作，不妨先读一读 Steve Blank 写的《初创公司领导手册》（*Startup Owner's Manual*，K&S Ranch 出版）。

在发布（发布对象也许是亲朋好友或此刻刚刚克隆代码仓库的内部人士）的瞬间你就会发现用户并不能通过航空公司、飞行日期、航班号这些信息决定所要获取的是哪一条记录。为了能真正利用这些数据，我们需要列表和检索的功能。

你可能也早就预料到了这一点。那么为什么还要发布这种有明显缺陷的半成品呢？这是因为尽管第二步要做什么很显而易见，但第十三步却还不明朗。我们应该在这一步就引入用户，因为他们对完成数据价值金字塔第一步起着关键作用。用户能够验证我们的前提假设，这一阶段的前提假设可能针对两类问题：“是否有人关注航班信息？”以及“对于航班信息来说，用户关心哪些内容？”我们对这些问题的回答分别是：“是”和“航班、出发地、目的地、飞机机尾编号、日期、飞行时间、飞行距离”。但是在未经验证前，我们没有十足的把握。如果没有与用户交互和学习，我们就是在黑暗中开发。这样下去基本不会成功，就好像没有牢固底座的金字塔很快就会坍塌一样。

应该现在发布的另一个原因在于，发布、展示并分享你的工作能够凸显平台配置等方面的一系列问题，而如果迟迟不发布，那么这些问题可能直到最终发布的时候还没被发现。在敏捷数据科学中，你应该保证在每个冲刺之后发布一个版本。作为团队成员，你无法决定是否正式交付，但是你能控制发布什么，向谁发布。对五个亲朋好友进行这种小发布是比较合适的，甚至还需要看着他们启动应用或者点开链接。在分享半成品应用时，你需要优化程序打包方式，解决依赖问题，你需要确保产品能看。如果不这样做，不为自己的奋斗设立一个清晰的产出，就会因为对产品太熟悉而发现不了一些隐匿的技术问题。

现在让我们增加航班记录列表功能，并扩展这一功能以支持搜索，这样我们就可以期待真实用户的访问了。

列出航班记录

按出发地和目的地筛选后，航班常常以按价格从低到高排序的列表呈现。我们没有价格数据，因此我们在把两个机场间的所有航班放入列表时，选择按起飞时间作为第一排序依据，降落时间作为第二排序依据进行排序。列表可以把相似的航班整理到一起。列表是数据价值金字塔本层级中展示单条记录的下一个步骤。

使用 MongoDB 列出航班记录

在我们开始搜索航班数据之前，我们需要能列出航班记录，以便展示搜索结果。我们可以使用 MongoDB 的查询功能来获取指定日期机场间的航班列表，并按照起飞降落时间排序。下面这条查询语句可以在 *ch04/mongo.js* 中找到：

```
$ mongo agile_data_science

>db.on_time_performance.find(
{Origin: 'ATL', Dest: 'SFO', FlightDate: '2015-01-01'}).sort(
{DepTime: 1, ArrTime: 1}) // 执行时间很长或不能正确执行
```

你会看到如下报错信息：

```
error: {
"$err" : "too much data for sort() with no index. add an index or specify a
smaller limit",
"code" : 10128
}
```

即使没有报错，这条语句也会花很久才能执行完，所以让我们再加上一个索引。一般来说，我们需要为应用中的每一种访问模式都加上索引，因此要始终牢记提前做好这些准备，以免将来受困于性能问题：

```
>db.on_time_performance.ensureIndex({Origin: 1, Dest: 1, FlightDate: 1})
```

现在我们有了起降地以及日期的索引，我们可以轻松获取 2015 年 1 月 1 日从 ATL 到 SFO 的所有航班：

```
>db.on_time_performance.find(
  {Origin: 'ATL', Dest: 'SFO',
   FlightDate: '2015-01-01'}).sort(
    {DepTime: 1, ArrTime: 1})
     // 很快执行完
```

我们的 Flask 部分也和以往一样，不过这次传递过来的是航班的数组而不是单个航班。这次我们要在网页控制器的 URL 中加上 slug。[1] slug 把参数放在斜杠之间，而不是使用查询参数。参考 *ch04/web/on_time_flask_template.py* 中的代码片段：

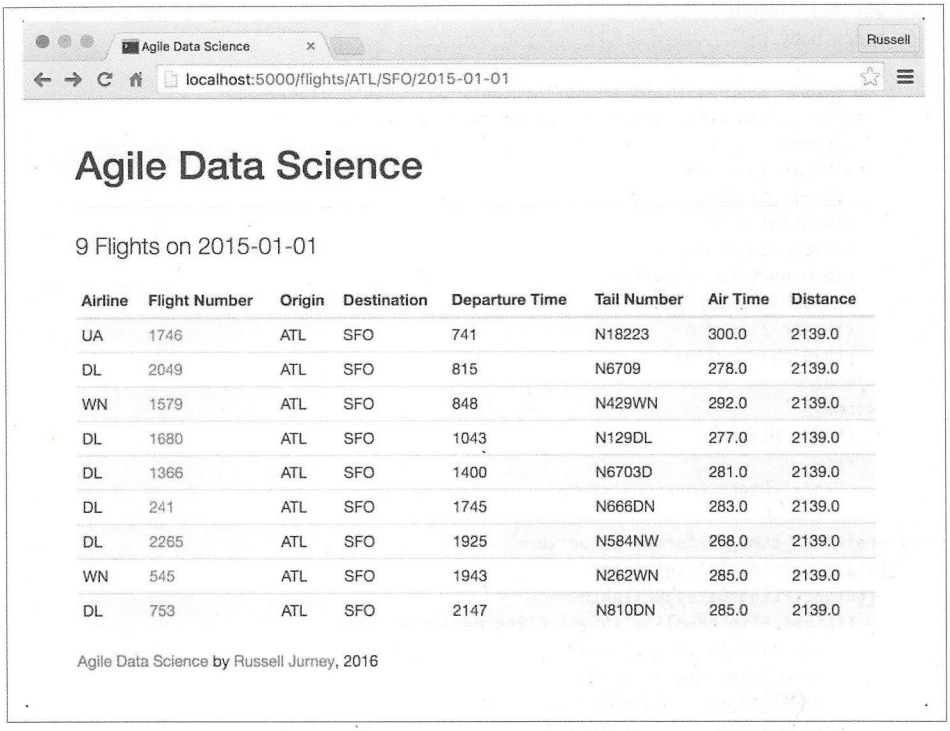
```
# 控制器：获取指定日期、两个城市之间的全部航班并展示
@app.route("/flights/<origin>/<dest>/<flight_date>")
def list_flights(origin, dest, flight_date):

  flights = client.agile_data_science.on_time_performance.find(
   {
     'Origin': origin,
     'Dest': dest,
     'FlightDate': flight_date
   },
   sort=[
     ('DepTime', 1),
     ('ArrTime', 1),
   ]
  )
  flight_count = flights.count()
  return render_template('flights.html', flights=flights,
   flight_date=flight_date, flight_count=flight_count)
```

Bootstrap 表的样式比较优雅，因此模板也变得很简单。表格经常被设计师嘲笑，但这是表格型的数据，因此用表格是再合适不过的。为了更加优雅，我们加上了当天的航班数量，并且由于日期对于所有记录是一样的，所以我们把它作为页面标题中的字段而不是一列：

1 欲了解更多控制器相关内容，请参考 Alex Coleman 的博文 *MVC in Flask*。——译者注

```
{% extends "layout.html" %}
{% block body %}
  <div>
    <p class="lead">{{flight_count}} Flights on {{flight_date}}</p>
    <table class="table table-condensed table-striped">
      <thead>
      <th>Airline</th>
      <th>Flight Number</th>
      <th>Origin</th>
      <th>Destination</th>
      <th>Departure Time</th>
      <th>Tail Number</th>
      <th>Air Time</th>
      <th>Distance</th>
    </thead>
    <tbody>
      {% for flight in flights %}
      <tr>
        <td>{{flight.Carrier}}</td>
        <td>
<a href="/on_time_performance?Carrier=
  {{flight.Carrier}}&FlightDate=
    {{flight.FlightDate}}&FlightNum=
      {{flight.FlightNum}}">{{flight.FlightNum}}</a></td>
          <td>{{flight.Origin}}</td>
          <td>{{flight.Dest}}</td>
          <td>{{flight.DepTime}}</td>
          <td>{{flight.TailNum}}</td>
          <td>{{flight.AirTime}}</td>
          <td>{{flight.Distance}}</td>
       </tr>
       {% endfor %}
      </tbody>
    </table>
  </div>
{% endblock %}
```

我们往一个模板中绑定许多值。注意，我们还加上了从列表页面（见图 4-6）到单条记录页面的链接，由航空公司、航班号、日期等构成。



Airline	Flight Number	Origin	Destination	Departure Time	Tail Number	Air Time	Distance
UA	1746	ATL	SFO	741	N18223	300.0	2139.0
DL	2049	ATL	SFO	815	N6709	278.0	2139.0
WN	1579	ATL	SFO	848	N429WN	292.0	2139.0
DL	1680	ATL	SFO	1043	N129DL	277.0	2139.0
DL	1366	ATL	SFO	1400	N6703D	281.0	2139.0
DL	241	ATL	SFO	1745	N666DN	283.0	2139.0
DL	2265	ATL	SFO	1925	N584NW	268.0	2139.0
WN	545	ATL	SFO	1943	N262WN	285.0	2139.0
DL	753	ATL	SFO	2147	N810DN	285.0	2139.0

图 4-6　展示航班列表

数据分页

现在我们可以列出指定日期、两个城市之间的航班，用户可能会说："如果我查出来的航班很多，浏览器会不会挂掉呢？"一次列出数百条记录确实看起来不太好，对于其他类型的数据来说也是一样的。难道不应该加上一组向前向后翻页的按钮吗？是的，这就是我们接下来要添加的功能。

到目前为止，我们已经基本讲完了如何创建模板、子模板和宏。下面我们将深入讲解如何利用宏和子模板实现分页。

重复造轮子

为什么要自己实现分页呢？这不是已经有解决方案的问题吗？

对于第一个问题，自己实现分页是把浏览器和数据直接连接起来的一个很好的实例。在敏捷数据科学中，我们尝试通过尽量少的操作建立后端到浏览器中图像的联系，把数据处理为用户屏幕上所处的状态。为什么呢？我们这样做是因为这可以降低我们系统的复杂度，

仅供非商业用途或交流学习使用

可以让数据科学家和设计师站在同样的立场上，而且这套哲学不依赖于表连接这类在超大型计算机或者传统系统中常用的技术，而是更契合分布式系统的天性。

在预测系统中，当模型复杂时，保持模型与视图一致是至关重要的。当围绕一个功能的交互设计能充分了解底层数据模型，并与其保持一致时，我们可以创造出最多的价值。数据科学家必须将对数据的理解传达给团队的其他成员，否则团队就无法达成共同的观点。建立一个匹配模型的视图的原则从一开始就确保了这一点。

在实践中，我们无法预测一个功能会出现在哪一层。它可能来自网页开发人员、设计师、数据科学家或者平台工程师的一次创意爆发。为了验证，我们应当尽快在实验版本中发布，这样这个功能的实现层可以在系统中任何一处随时展开。当发生这种情况时，我们必须进行记录，并标记它为包含技术债的功能。在功能稳定之后，如果还要保留在系统中，我们会在时间允许的情况下把它放回正式系统中。

像 Rails 或 Django 这样的完整的应用程序框架可能会内建了这个功能。然而，当我们在派生数据的基础上构建应用程序时，交互机制可能变化很小，也可能天翻地覆。大多数 Web 框架都是围绕 CRUD 操作进行优化的。在大数据的探索和可视化中，我们只做 CRUD 的读取部分，而且还要把其中的可视化做得相对复杂。在这种情况下，框架的用处就不那么大了，因为我们要自定义很多行为。还要注意的是，虽然 MongoDB 恰好有选择和返回一系列排好序的记录的功能，但是你使用的 NoSQL 存储可能提供也可能不提供此功能，甚至可能因为在数据不断发布时需要进一步处理而无法使用此功能。你可能需要定期预计算数据，并自行提供数据列表。NoSQL 为我们提供了另一种可能的选择，而 Web 框架则是针对关系型数据库进行优化的。我们必须时刻把这些事情掌握在自己手中。

提供分页的数据

在开始之前，我们要先改造控制器，使之适用 MongoDB 的分页功能（*https://api.mongodb.com/python/current/api/pymongo/cursor.html*）。这里需要一点点计算，毕竟 MongoDB 的分页是通过 `skip` 和 `limit` 来实现的，而不是直接使用 `start` 和 `end`。我们需要将 `end` 减去 `start` 的差值作为查询结果所要的条数，然后以它为参数调用 `limit`：

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

```
# 控制器：获取指定日期、两个城市之间的全部航班并展示
@app.route("/flights/<origin>/<dest>/<flight_date>")
def list_flights(origin, dest, flight_date):

  start = request.args.get('start') or 0
  start = max(int(start) - 1, 0)
  end = request.args.get('end') or 20
  end = int(end)
  width = end - start

flights = client.agile_data_science.on_time_performance.find(
 {
   'Origin': origin,
   'Dest': dest,
   'FlightDate': flight_date
  },
  sort=[
    ('DepTime', 1),
    ('ArrTime', 1),
  ]
).skip(start).limit(width)
flight_count = flights.count()
return render_template('flights.html', flights=flights,
 flight_date=flight_date, flight_count=flight_count)
```

从 HTML 提炼原型

为了在模板中实现分页，我们需要从 HTML 中提炼原型。我们都对在航空公司网站上浏览航班时的下一页 / 上一页按钮很熟悉，我们要为列出航班提供同样的功能。我们要在航班列表页面（见图 4-7）的底部增加链接，可以进入前一页或者后一页。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

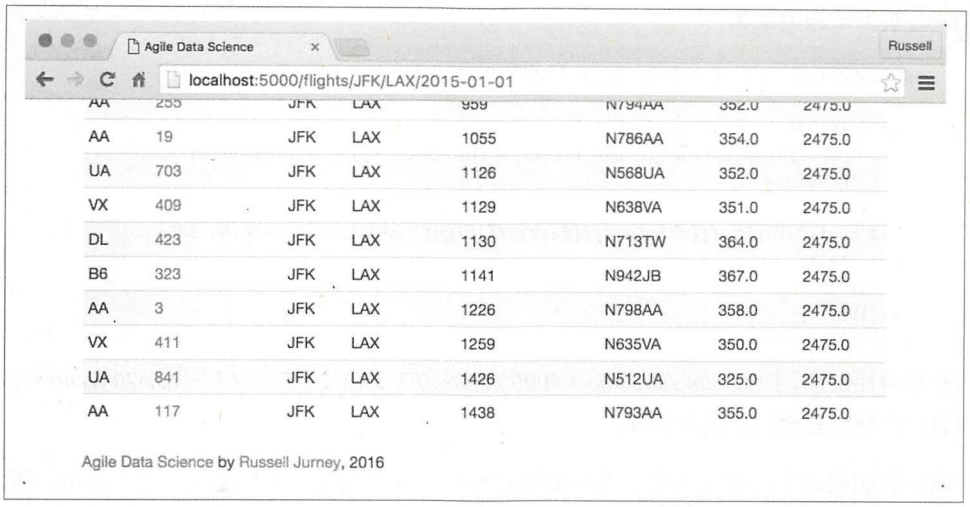


图 4-7　缺失下一页 / 上一页链接

更具体地说，我们需要能增减路径 */flights/<origin>/<dest>/<date>?start=N&end=N* 中的偏移范围的链接。我们先根据这些需求，以我们列出的航班 API 做出静态的前翻后翻链接（见图 4-8）作为原型。

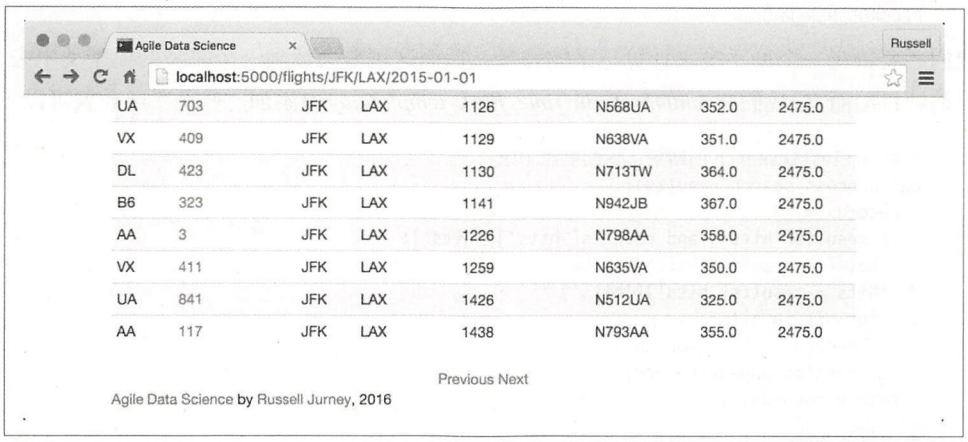


图 4-8　简易的下一页 / 上一页链接

例如，对于 URL 地址 */flights/JFK/LAX/2015-01-01?start=20&end=40*，我们想要把这个

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

HTML 页面动态渲染为

```
# /ch04/templates/partials/flights.html
...
  <div style="text-align: center">
   <a href="/flights/{{origin}}/{{dest}}/{{flight_date}}?start=0&end=20">
       Previous
   </a>
   <a href="/flights/{{origin}}/{{dest}}/{{flight_date}}?start=40&end=60">
       Next
   </a>
  </div>
{% endblock -%}
```

粘贴并前往链接，比如 *http://localhost:5000/flights/JFK/LAX/2015-01-01?start=20 &end=40*，用我们的数据验证链接是否正确。

下面让我们把这个过程通用化。直接使用宏很方便，但是我们不想让模板过于复杂，所以我们把增量的计算放到 Python 的一个助手类（我们可能需要一个模型（model）类）中，然后使用宏渲染得到的偏移量。

对于初学者来说，可以利用这个机会学习增加一个简易的配置文件，在里面设置一些变量，比如，每页展示的记录条数（把这些都放在代码里面会导致以后有很多麻烦事）：

```
# ch04/web/config.py, index.py 的配置文件
RECORDS_PER_PAGE = 20
```

我们还要创建一个助手类来计算记录偏移量。后面这可能会放到一个完整的模型类中去，但是就目前而言，我们在 */ch04/web/on_time_flask_template.py* 中添加一些助手函数就可以了：

```
# 处理 Elasticsearch 的结果，返回航班记录
def process_search(results):
  records = []
  if results['hits'] and results['hits']['hits']:
    total = results['hits']['total']
    hits = results['hits']['hits']
    for hit in hits:
      record = hit['_source']
      records.append(record)
  return records, total

# 计算从 MongoDB 中获取航班列表的偏移量
def get_navigation_offsets(offset1, offset2, increment):
  offsets = {}
  offsets['Next'] = {'top_offset': offset2 + increment, 'bottom_offset':
  offset1 + increment}
  offsets['Previous'] = {'top_offset': max(offset2 - increment, 0),
'bottom_offset': max(offset1 - increment, 0)} # 不让它小于 0
 return offsets
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

```
# 从查询字符串中去除 start 和 end 参数
def strip_place(url):
  try:
    p = re.match('(.+)&start=.+&end=.+', url).group(1)
  except AttributeError, e:
    return url
  return p
```

这个控制器现在可以使用助手来生成导航变量并把变量绑定到模板中了，因为我们获取了航班的列表以及导航链接所要计算的偏移量，如 *ch04/web/ on_time_flask_template.py* 所示：

```
# 控制器：获取指定日期、两个城市之间的全部航班并展示
@app.route("/flights/<origin>/<dest>/<flight_date>")
def list_flights(origin, dest, flight_date):

  start = request.args.get('start') or 0
  start = int(start)
  end = request.args.get('end') or 20
  end = int(end)
  width = end - start

  nav_offsets = get_navigation_offsets(start, end, config.RECORDS_PER_PAGE)

  flights = client.agile_data_science.on_time_performance.find(
   {
     'Origin': origin,
     'Dest': dest,
     'FlightDate': flight_date
   },
   sort=[
     ('DepTime', 1),
     ('ArrTime', 1),
   ]
)
flight_count = flights.count()
flights = flights.skip(start).limit(width)

return render_template(
  'flights.html',
  flights=flights,
  flight_date=flight_date,
  flight_count=flight_count,
  nav_path=request.path,
  nav_offsets=nav_offsets

  )
```

我们的航班列表模板 *ch04/web/templates/flights.html* 会调用宏来渲染数据。注意，使用 `|safe` 确保 HTML 没有被转义：

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

```
{% import "macros.jnj" as common %}
{% if nav_offsets and nav_path -%}
  {{ common.display_nav(nav_offsets, nav_path, flight_count, query)|safe }}
{% endif -%}
```

我们把这段代码放在 Jinja2 宏文件中，把任务进一步细化为在 div 中加两个链接：

```
ch04/web/templates/macros.jnj
<!-- 在航班列表中展示两个用于前一页 / 后一页的导航链接 -->
{% macro display_nav(offsets, path, count, query) -%}
  <div style="text-align: center;">
    {% for key, values in offsets.items() -%}
      {%- if values['bottom_offset'] >= 0 and values['top_offset'] >
       0 and count > values['bottom_offset'] -%}
        <a style="margin-left: 20px; margin-right: 20px;"
          href="{{ path }}?start={{ values
          ['bottom_offset'] }}&end={{ values['top_offset']
           }}{%- if query -%}?search=
          {{query}}{%- endif -%}">{{ key }}</a>
      {% else -%}
        {{ key }}
      {% endif %}
    {% endfor -%}
  </div>
{% endmacro -%}
```

这样就搞定了。我们现在可以像在别的航班网站上一样分页浏览我们的航班列表了。我们距离真正能让用户使用的状态又近了一步，还有了到单个航班记录页面的链接的拓扑结构。随着我们继续攀登数据价值金字塔，这一结构可以支持后面更多的结构。

搜索航班数据

浏览航班列表确实比手动查找 message_ids 要方便，但它在搜索感兴趣的航班时并没有那么高效。让我们为我们的数据平台添加搜索功能。

创建索引

在我们把文档存储到 Elasticsearch 中之前，我们需要先创建索引。请参照 *elastic_scripts/create.sh*。注意，我们只配置了一个分片（shard）和一份数据（replica）。在生产环境中，你需要把系统配置在多台机器组成的集群上并配置多个分片，而为了实现冗余和高可用性，每个分片也需要配置多份备份。对于这个情况，分片和数据重复均设置为 1 就足够了！

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

```
#!/usr/bin/env bash
curl -XPUT 'http://localhost:9200/agile_data_science/' -d '{
    "settings" : {
        "index" : {
            "number_of_shards" : 1,
            "number_of_replicas" : 1
        }
    }
}'
```

直接运行这个脚本，接下来我们就可以把准点数据记录发布到 Elasticsearch 中了：

```
elastic_scripts/create.sh
```

注意，如果你要重来一遍，你可以用下面这条命令删掉 `agile_data_science` 索引：

```
elastic_scripts/drop.sh
```

和创建索引的脚本一样，这个脚本也会调用 `curl`：

```
#!/usr/bin/env bash

curl -XDELETE 'http://localhost:9200/agile_data_science/'
```

发布航班数据到 Elasticsearch

使用第 2 章中创建的方式，我们可以把航班准点情况数据轻松写入 Elasticsearch，如 *ch04/pyspark_to_elasticsearch.py* 所示：

```
# 读取 Parquet 文件
on_time_dataframe = spark.read.parquet('data/on_time_performance.parquet')

# 将 DataFrame 写入 Elasticsearch
on_time_dataframe.write.format("org.elasticsearch.spark.sql")\
  .option("es.resource","agile_data_science/on_time_performance")\
  .option("es.batch.size.entries","100")\
  .mode("overwrite")\
  .save()
```

注意，我们需要把 `es.batch.size.entries` 设为 100，而默认值为 1000。这使得 Elasticsearch 不会被 Spark 压垮。你可以在配置指南中找到更多可以调节的设置项。

注意，这一过程可能需要花费一些时间，毕竟要把几百万条记录添加到索引里。你可能需要等待执行一段时间。你也可以中断执行，然后继续后面的步骤；只要已经索引了一部分记录，应该就可以。同样地，如果执行中发生了报错，你可以通过下面的查询结果进行检查；有可能忽略这些错误直接进入后面的步骤，只要已经写入了足够多的记录以供本例后续使用即可。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

用 curl 查询数据很简单。亚特兰大（机场代码 ATL）是全世界最繁忙的机场，我们以从那里起飞的航班为例：

```
curl \
  'localhost:9200/agile_data_science/on_time_performance/ \
    _search?q=Origin:ATL&pretty'
```

输出如下：

```
{
  "took": 7,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
},
"hits": {
  "total": 379424,
  "max_score": 3.7330098,
  "hits": [
   {
      "_index": "agile_data_science",
      "_type": "on_time_performance",
      "_id": "AVakkdOGX8o-akD569e2",
      "_score": 3.7330098,
      "_source": {
        "Year": 2015,
        "Quarter": 3,
        "Month": 9,
        "DayofMonth": 23,
        "DayOfWeek": 3,
        "FlightDate": "2015-09-23",
        "UniqueCarrier": "DL",
        ...
```

查询的返回结果包含查询得到的记录以及其他一些有用的信息，包括查询的执行时间（7 ms）以及符合查询条件的记录总数（379,424）。

通过网页搜索航班数据

下面，让我们把我们的搜索引擎与网页关联起来。

首先，配置 pyelastic 指向 Elasticsearch 服务器：

```
# ch04/web/config.py
ELASTIC_URL = 'http://localhost:9200/agile_data_science'
```

仅供非商业用途或交流学习使用

接下来，使用 *ch04/web/on_time_flask_template.py* 导入、配置、查询 Elasticsearch，对应到 */flights/search* 路径：

```
@app.route("/flights/search")
def search_flights():

  # 查询参数
  carrier = request.args.get('Carrier')
  flight_date = request.args.get('FlightDate')
  origin = request.args.get('Origin')
  dest = request.args.get('Dest')
  tail_number = request.args.get('TailNum')
  flight_number = request.args.get('FlightNum')

  # 分页参数
  start = request.args.get('start') or 0
  start = int(start)
  end = request.args.get('end') or config.RECORDS_PER_PAGE
  end = int(end)

  nav_offsets = get_navigation_offsets(start, end, config.RECORDS_PER_PAGE)

  # 构造 Elasticsearch 查询
  query = {
    'query': {
      'bool': {
        'must': []}
    },
    'sort': [
      {'FlightDate': {'order': 'asc', 'ignore_unmapped' : True} },
      {'DepTime': {'order': 'asc', 'ignore_unmapped' : True} },
      {'Carrier': {'order': 'asc', 'ignore_unmapped' : True} },
      {'FlightNum': {'order': 'asc', 'ignore_unmapped' : True} },
      '_score'
    ],
    'from': start,
    'size': config.RECORDS_PER_PAGE
  }
  if carrier:
    query['query']['bool']['must'].append({'match': {'Carrier': carrier}})
  if flight_date:
    query['query']['bool']['must'].append({'match': {'FlightDate': flight_date}})
  if origin:
    query['query']['bool']['must'].append({'match': {'Origin': origin}})
  if dest:
    query['query']['bool']['must'].append({'match': {'Dest': dest}})
  if tail_number:
    query['query']['bool']['must'].append({'match': {'TailNum': tail_number}})
  if flight_number:
    query['query']['bool']['must'].append(\n||||{...}\n||||) # where |
                                                             # is a space
```

仅供非商业用途或交流学习使用

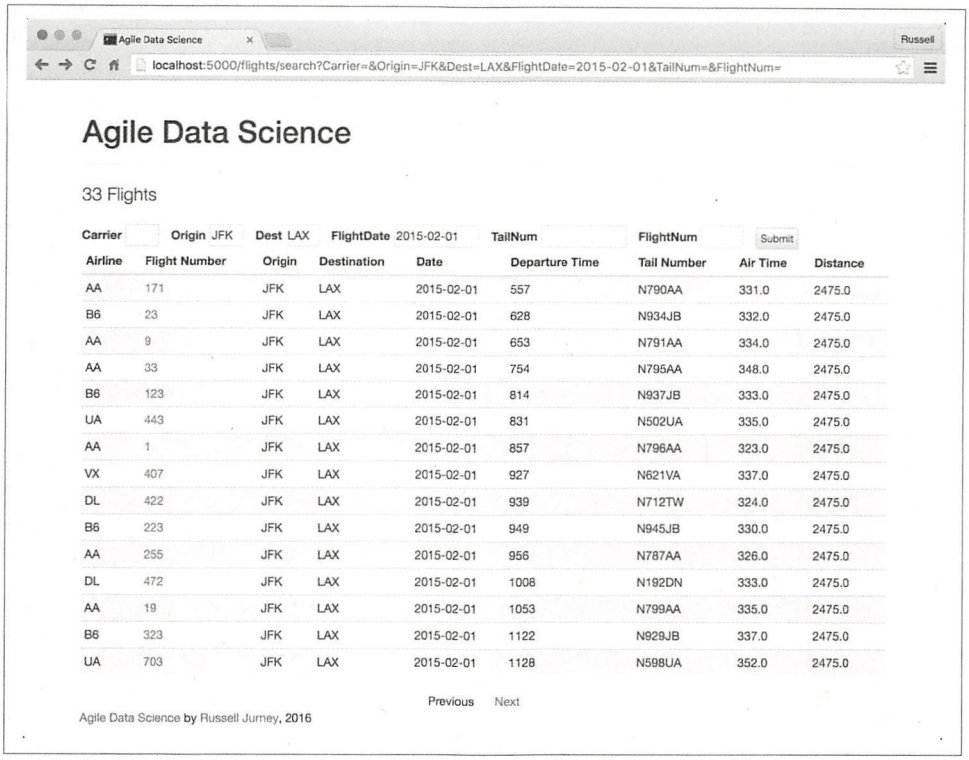
```
    results = elastic.search(query)
    flights, flight_count = process_search(results)

    # 将搜索参数存在表单模板中
    return render_template(
      'search.html',
      flights=flights,
      flight_date=flight_date,
      flight_count=flight_count,
      nav_path=request.path,
      nav_offsets=nav_offsets,
      carrier=carrier,
      origin=origin,
      dest=dest,
      tail_number=tail_number,
      flight_number=flight_number
      )
```

导航链接生成已经通用化了，我们可以用类似的模板也分页显示航班搜索的结果。我们需要一个表单来根据特定字段搜索航班，所以在页面最上面写了一个表单。我们也把所有搜索参数预填在返回页面的表单中，方便我们下次提交查询时使用上次的部分参数。如 *ch04/web/templates/search.html* 所示：

```
<form action="/flights/search" method="get">
  <label for="Carrier">Carrier</label>
  <input name="Carrier" maxlength="3" style="width: 40px; margin-right: 10px;"
      value="{{carrier}}"></input>
  <label for="Origin">Origin</label>
  <input name="Origin" maxlength="3" style="width: 40px; margin-right: 10px;"
      value="{{origin}}"></input>
  <label for="Dest">Dest</label>
  <input name="Dest" maxlength="3" style="width: 40px; margin-right: 10px;"
      value="{{dest}}"></input>
  <label for="FlightDate">FlightDate</label>
  <input name="FlightDate" style="width: 100px; margin-right: 10px;"
      value="{{flight_date}}"></input>
  <label for="TailNum">TailNum</label>
  <input name="TailNum" style="width: 100px; margin-right: 10px;"
      value="{{tail_number}}"></input>
  <label for="FlightNum">FlightNum</label>
  <input name="FlightNum" style="width: 50px; margin-right: 10px;"
      value="{{flight_number}}"></input>
  <button type="submit" class="btn btn-xs btn-default" style="height: 25px”>
   Submit
  </button>
</form>
```

搜索航班的情况见图 4-9。

仅供非商业用途或交流学习使用



Airline	Flight Number	Origin	Destination	Date	Departure Time	Tail Number	Air Time	Distance
AA	171	JFK	LAX	2015-02-01	557	N790AA	331.0	2475.0
B6	23	JFK	LAX	2015-02-01	628	N934JB	332.0	2475.0
AA	9	JFK	LAX	2015-02-01	653	N791AA	334.0	2475.0
AA	33	JFK	LAX	2015-02-01	754	N795AA	348.0	2475.0
B6	123	JFK	LAX	2015-02-01	814	N937JB	333.0	2475.0
UA	443	JFK	LAX	2015-02-01	831	N502UA	335.0	2475.0
AA	1	JFK	LAX	2015-02-01	857	N796AA	323.0	2475.0
VX	407	JFK	LAX	2015-02-01	927	N621VA	337.0	2475.0
DL	422	JFK	LAX	2015-02-01	939	N712TW	324.0	2475.0
B6	223	JFK	LAX	2015-02-01	949	N945JB	330.0	2475.0
AA	255	JFK	LAX	2015-02-01	956	N787AA	326.0	2475.0
DL	472	JFK	LAX	2015-02-01	1008	N192DN	333.0	2475.0
AA	19	JFK	LAX	2015-02-01	1053	N799AA	335.0	2475.0
B6	323	JFK	LAX	2015-02-01	1122	N929JB	337.0	2475.0
UA	703	JFK	LAX	2015-02-01	1128	N598UA	352.0	2475.0

图 4–9　搜索航班的情况

本章小结

我们已经学习了对航班记录的收集、发布、索引、展示、列表、搜索。这些记录对我们来说不再是抽象的概念。我们可以搜索航班，单击某一个，也可以用同样方式探索其他的数据集。更重要的是，我们学会了将通过我们的平台转化后的原始数据连接到交互式应用中。

这个应用构成了我们价值栈的基础。我们要在全书中用它来开发、展示和遍历更高级的特性，在我们攀登数据价值金字塔的路上创造价值。有了金字塔的基础，我们就可以开始构建图表了。

第 5 章

使用图表进行数据可视化

接下来是我们的第二个敏捷开发冲刺，要把数据转为图表（见图 5-1）。

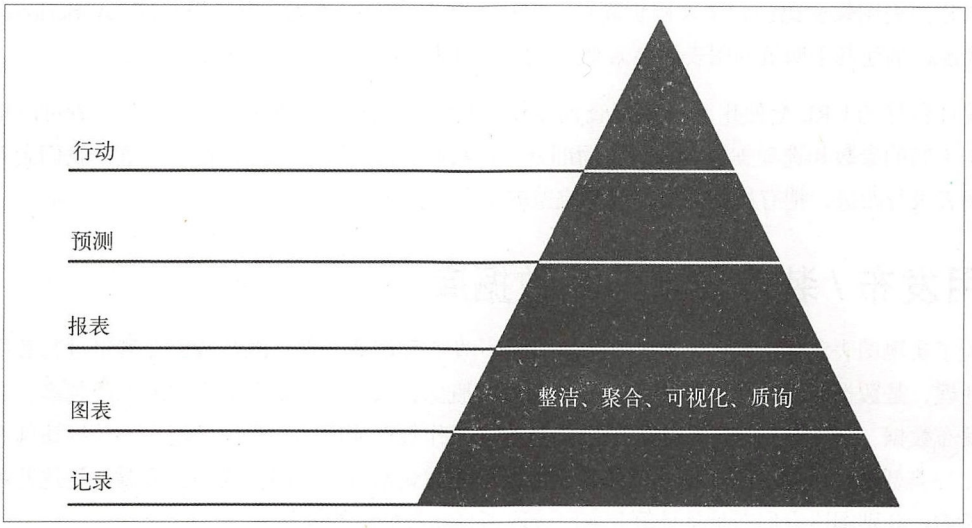


图 5–1　第二层：使用图表进行数据可视化

图表是我们对汇总数据的第一次观察，把多条记录的属性映射到视觉形式的表示中，这样有助于我们理解并纵览数据。我们在这一步中的目标是发布图表以引起用户对数据的兴趣，让用户与数据交互，构建可重用的有助于在下一步骤中交互式探索数据的工具，并且开始提炼数据的结构与实体，这样我们可以使用这一结构创建新的功能与见解。

本章的代码示例可以在 *Agile_Data_Code_2/ch05* 中找到。克隆代码仓库，跟上我们！

```
git clone https://github.com/rjurney/Agile_Data_Code_2.git
```

图表质量：迭代至关重要

一个好的图表应该会讲故事，用户可以从图表中得出见解，可以发现有趣的内容并分享和做出应对。实际上，大多数图表并不能实现这样的目标，几乎没什么价值。能讲故事的图表非常罕见，因为大多数人做出图表后就结束了。实际上，你必须迭代改进图表才能实现有用的可视化。做好在得到几个有用的图表前要丢弃很多图表的心理预期，不要试图提前定好图表细节，否则你会失望。使用你的直觉和好奇心，在即席的交互式探索式数据分析中有机地添加图表。

你可以先以即席查询的方式构建图表，随着工作不断展开，工作流程应该越来越自动化和可重复。在敏捷数据科学中，我们优先通过网页进行可视化。尽管使用 matplotlib 或者 R 创建图表图像要比创建图表网页简单，但这种状况在迅速改变。有了现代的 JavaScript 图表库，创建基于网页的图表不会太难，因此我们从一开始就使用它来做可视化。

设计良好的 URL 会使用 slug 或者查询参数，我们可以进行归纳从而使得一张图表可以根据不同的参数和选项支持数据的不同部分。在我们掌握了图表之后，在下一章中我们会对图表进行改进，把有用的图表扩展为完整的交互式报表。

用发布 / 装饰模型伸缩数据库

为了实现图表和其他服务，我们要怎样计算、发布和消费数据，以及我们在哪里进行数据处理，是数据库规范化过程中我们必须理解的概念。我们在批量计算中做的处理越多，在发布数据、读取数据库一层中做的处理越少，操作数据库所需的时间就越少。数据访问有几种典型模式，相应的数据处理和数据库操作的方式都有所不同。我们会简要讨论这几种模式，分别阐述它们在批量计算和实时读取所发布数据中的操作。

哪种形式适合你，取决于你的应用和数据访问模式、硬件预算，以及你希望依赖、操作和调优数据库的程度。我们批处理做得越少，使用的存储形式效率越低，依赖的数据库功能越多，数据库需要的调优和运维工作就越多。如果我们喜欢运维数据库，这可能挺好。否则，在大多数时候来看，这并不好，毕竟我不想每周花几分钟以上的时间去运维数据库。

我们将按伸缩性从高到低，所需数据库系统复杂性从低到高的顺序讨论如何存储时间序列图表。比如，运维像 Cassandra 或者 Voldemort 这样的键 / 值对存储系统就极其简单。由

于单点故障不会导致系统崩溃，故你可以放心睡觉，不用盯着数据库。相反，如果使用 Bigtable 或者类似的有主节点概念的系统，单点故障可能会导致整个系统崩溃，系统崩溃发生的可能性更高了。在主节点崩溃的时候，应用至少有一段时间无法运行。这需要管理任务进行修复。而如果你使用了像 MongoDB 或 MySQL 这样功能更丰富的数据库，你就必须管理你使用的那些功能，比如，创建索引来实现更高效的访问。

不过需要注意的是，不论数据库提供了多少功能，任何数据库都可以仅仅当作键 / 值对存储系统来使用。MySQL 功能很丰富，但是如果你不用它的那些功能，它就成了一个简单而高效的键 / 值对存储系统。你可以直接把数据的 JSON 表示存储进去，通过主键进行查询访问。如果你是 MySQL 专家，这么做可能也挺好的。

我想说明的是，批量计算的工作越多且使用数据库的功能越少，操作应用就会越简单，伸缩性也就越好。使用数据库功能要三思。倾向于使用批量计算而不是使用数据库功能，你就可以构建出一个能自己运转良好的应用，而不用一直劳心费力。

一阶形式

数据伸缩性最好的形式是把图表、表格、预测模型都作为整体提前准备好，作为单个嵌套对象存储到键 / 值对存储系统或者文档存储系统中（不过从技术上来讲，你可以把数据以主键分组后用 JSON 格式编码，把任何数据库当作键 / 值对存储系统来用）。例如，要存储一个时间序列图表的数据，我们可以计算出图表中数据排序后的列表，把这个列表放到一个对象中并分配主键，这样我们就可以通过一条查询语句获取这个图表的数据了。

图 5-2 展示了这种准备 / 发布模型。在这种形式中，我们准备好一个对象，其中字段 `Flights` 包含排好序的航班列表。这个对象的主键是 `TailNum`，我们可以用这个字段来访问数据。

仅供非商业用途或交流学习使用

```
{
    "TailNum" : "N16954"
    "Flights" : [
        [
            "EV",
            "2015-01-07",
            3926,
            "IAH",
            "MFE"
        ],
        ...
    ]
}
```

图 5-2　为文档存储系统准备的对象

这种形式是伸缩性最好的，因为我们只需要用键 / 值对存储系统来存储数据，要查询数据则只需要用主键。从键 / 值对存储系统中直接读取数据要比从关系型数据库读取简单很多。系统运维也会简单很多，因为不用担心单点故障导致的系统崩溃，执行的也都是简单的操作。记住，一阶形式让操作变得简单。

二阶形式

二阶形式的伸缩性仅次于一阶形式。它利用了 *Google Bigtable*（*https://hbase.apache.org/book.html#scan*）及 *Apache HBase* 等衍生项目中键范围查询（*http://hbase.apache.org/*）的功能。HBase 表中的数据按键的字典序排序存储。这是很关键的功能，它让我们可以高效访问一定范围内的数据，因为键相似的记录都在磁盘上按顺序存储。

通过设计主键，我们可以实现很多原本要用关系型数据库来实现的操作。*Apache Phoenix*（*http://phoenix.apache.org/*）基于 HBase 提供了 SQL 抽象，这样你不需要写 Java 代码就能操作 HBase 了。Phoenix 是（需要快速解决问题的）应用开发人员使用 HBase 的好帮手。如果你想深入了解 HBase，可以读一读 Amandeep Khurana 写的“HBase 结构设计简介”。

仅供非商业用途或交流学习使用

我们在本书中没有使用 HBase，不过图 5-3 展示了上一个例子用 HBase 实现的样子。要实现等效的查询，我们需要为我们的数据组合构建一个主键，这样在我们把记录按照这个主键排序存储后，当对 `TailNum` 查询时，可以产生一个排好序的航班列表，和上一个例子中的结果完全一样。

Key: TailNum + FlightDate + FlightNum + Origin + Dest	Carrier	FlightDate	FlightNum	Origin	Dest	TailNum
N16953-2015-12-31-1241-ATL-SFO	DL	2015-12-31	1241	ATL	SFO	N16953
N16954-2015-01-07-3926-IAH-MFE	EV	2015-01-07	3926	IAH	MFE	N16954
N16954-2015-01-07-3926-MFE-IAH	EV	2015-01-07	3926	MFE	IAH	N16954
...						
N16955-2015-01-01-1611-LAX-LAS	AA	2015-01-01	1611	LAX	LAS	N16955

SCAN

图 5-3　存储在 HBase 中的文档，使用复合键利用范围查询的特性

通过新组合的键，我们可以通过范围查询实现对很多种查询的处理。这个功能是相当强大的，而且还具有相当好的伸缩能力。有一些 HBase 应用可以轻松处理 PB 级的数据。我们可以很容易地使用 HBase 和 Apache Phoenix 实现本书中的许多示例。记住，如果不知道如何用一阶形式，二阶形式也是不错的选择。

三阶形式

前两种形式之后，最高效的数据存储形式是把按时间或者分类归纳后的数据存储在 MySQL 或者 MongoDB 这样的数据库系统中，利用系统提供的 B 树（*https://en.wikipedia.org/wiki/B-tree*）索引（*https://en.wikipedia.org/wiki/Database_index*）高效查询表中的一部分记录，或是进行表连接。这有可能用到范围查询，也可能用到一些更复杂的随机查询，与对排好序的表进行范围查询完全不同。这些数据库通常有 GROUP BY 的功能，可以进行聚合计算，和我们用 Spark SQL 所实现的差不多。

查询功能丰富的关系型文档存储系统可以在查询时立刻计算出各种值，不然的话就要使用一种混合的策略，我们需要选出一定范围内的记录并预先归纳计算，和键范围查询类似。我们在本书中不会展示这种三阶形式，不过你可能已经对这种形式比较熟悉了。它会打破我们在本书中使用的发布 / 装饰模型，要不然就是仅仅提供了对预先聚合好的数据的范围查询。

选择一种形式

总之，选择低阶的形式可以让系统更容易伸缩，甚至实现水平伸缩。但是请记住，你可以选择功能更丰富的数据库，然后除迫切需要外不使用那些功能。比如，你可以使用 MySQL

仅供非商业用途或交流学习使用

作为键 / 值对存储系统，并为提前聚合好的数据提供范围查询。这样使用的话 MySQL 也能轻松伸缩。不过，在需要的时候，你也可以用 GROUP BY 功能来实现应用的新功能。如果新功能很受欢迎，再随时把相应的操作移回我们软件栈的批处理层中。

这里关键的经验就是*使用数据库功能时要三思*，因为这些功能用得越多，应用就越难扩容。实现批量计算和让一个庞大而功能丰富的数据库实例在高负载下运转相比要相对简单。你需要充分了解数据库，理解系统每个功能的使用会对整体性能带来的后果。

探究时令性

我们要寻找切入点，让我们从一个问题展开：一年中哪个月飞机出行最繁忙？

这个问题涉及时令（*seasonality*）。当一个计量值随一年中时间不断循环变化时，时令性就出现了。比如，圣诞灯的销售就有很强的时令性（尽管我全年都挂着那些灯），在每年的十二月销量都会达到一个极大值。

我们正好可以借此机会展示 SQL 和 NoSQL 数据流如何共同工作并实现互补。让我们制作我们的第一个图表，计算 2015 年每个月的航班总数。用 SQL 表达按月统计航班是很容易的。

查询并展示航班总数

```
PYSPARK_DRIVER_PYTHON=ipython pyspark
```

PySpark 脚本 *ch05/total_flights.py* 如下所示：

```
# 读取 Parquet 文件
on_time_dataframe = spark.read.parquet('data/on_time_performance.parquet')

# 使用 SQL 查询 2015 年每个月的航班总数
on_time_dataframe.registerTempTable("on_time_dataframe")
total_flights_by_month = spark.sql(
  """SELECT Month, Year, COUNT(*) AS total_flights
  FROM on_time_dataframe
  GROUP BY Year, Month
  ORDER BY Year, Month"""
)

# map/asDict 操作是让输出的记录更美观，但不是必需的
flights_chart_data = total_flights_by_month.rdd.map(lambda row: row.asDict())
flights_chart_data.collect()
```


仅供非商业用途或交流学习使用

这样我们就得到了图表要用的原始数据：

```
[{'Month': 1, 'Year': 2015, 'total_flights': 469968},
 {'Month': 2, 'Year': 2015, 'total_flights': 429191},
 {'Month': 3, 'Year': 2015, 'total_flights': 504312},
 {'Month': 4, 'Year': 2015, 'total_flights': 485151},
 {'Month': 5, 'Year': 2015, 'total_flights': 496993},
 {'Month': 6, 'Year': 2015, 'total_flights': 503897},
 {'Month': 7, 'Year': 2015, 'total_flights': 520718},
 {'Month': 8, 'Year': 2015, 'total_flights': 510536},
 {'Month': 9, 'Year': 2015, 'total_flights': 464946},
 {'Month': 10, 'Year': 2015, 'total_flights': 486165},
 {'Month': 11, 'Year': 2015, 'total_flights': 467972},
 {'Month': 12, 'Year': 2015, 'total_flights': 479230}]
```

保存到 MongoDB：

```
# 保存图表数据到 MongoDB
import pymongo_spark
pymongo_spark.activate()
flights_chart_data.saveToMongoDB('mongodb://localhost:27017/ \
agile_data_science.flights_by_month')
```

验证保存成功：

```
>db.flights_by_month.find().sort({"Year": 1, "Month": 1})
{ "_id": ObjectId(
  "56ff1246b050473d23777138"
),
  "total_flights": 469968,
  "Month": 1,
  "Year": 2015
}
{
  "_id": ObjectId("56ff1246b050473d23777134"),
  "total_flights": 429191,
  "Month": 2,
  "Year": 2015
}
{
  "_id": ObjectId("56ff1246b050473d23777137"),
  "total_flights": 504312,
  "Month": 3,
  "Year": 2015
}
{
  "_id": ObjectId("56ff1246b050473d2377713a"),
  "total_flights": 485151,
  "Month": 4,
  "Year": 2015
}
...
```


仅供非商业用途或交流学习使用

接下来创建一个新的 Flask 控制器给图表 HTML 页面提供 JSON 格式的数据：

```
# 控制器：获取航班图表
@app.route("/total_flights")
def total_flights():
  total_flights = client.agile_data_science.flights_by_month.find({},
    sort=[
      ('Year', 1),
      ('Month', 1)
    ])
return render_template('total_flights.html', total_flights=total_flights)

# 通过异步请求提供图表数据（曾被称为 AJAX）
@app.route("/total_flights.json")
def total_flights_json():
  total_flights = client.agile_data_science.flights_by_month.find({},
    sort=[
      ('Year', 1),
      ('Month', 1)
    ])
  return json_util.dumps(total_flights, ensure_ascii=False)
```

注意，在我们制作图表前，让我们先用上一章中的方法创建一个简单的表格：

```
{% extends "layout.html" %}
{% block body %}
  <div>
    <p class="lead">Total Flights by Month</p>
    <table class="table table-condensed table-striped"style="width: 200px;">
      <thead>
        <th>Month</th>
        <th>Total Flights</th>
      </thead>
      <tbody>
        {% for month in total_flights %}
        <tr>
          <td>{{month.Month}}</td>
          <td>{{month.total_flights}}</td>
        </tr>
        {% endfor %}
      </tbody>
    </table>
  </div>
{% endblock %}
```

每个月的航班总数见图 5-4。

仅供非商业用途或交流学习使用

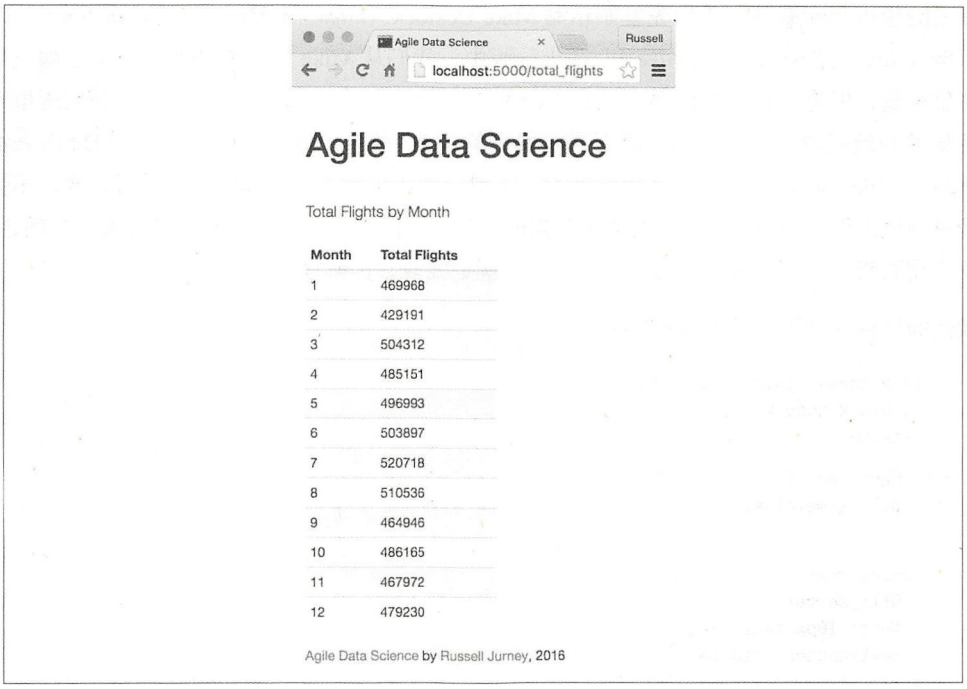


图 5-4　每个月的航班总数

这就完成了表格数据展示！这只是起步，我无法看着表就发现数据的趋势，对吧？让我们用 D3 做一个时间序列图表。我们先用 Flask 做一个提供图表页面的控制器，然后派生出能为我们的图表提供 JSON 格式数据的控制器：

```
# 控制器：获取航班图表
@app.route("/total_flights_chart")
def total_flights_chart():
  total_flights = client.agile_data_science.flights_by_month.find({},
    sort=[
      ('Year', 1),
      ('Month', 1)
    ])
  return render_template('total_flights_chart.html', total_flights=total_flights)

# 通过异步请求提供图表数据（曾被称为 AJAX）
@app.route("/total_flights.json")
def total_flights_json():
  total_flights = client.agile_data_science.flights_by_month.find({},
    sort=[
      ('Year', 1),
      ('Month', 1)
    ])
  return json_util.dumps(total_flights, ensure_ascii=False)
```


仅供非商业用途或交流学习使用

图表的模板相对简单：我们先复制粘贴 Mike Bostock（*https://bost.ocks.org/mike/bar/3/*）的示例（*https://bost.ocks.org/mike/*）。在上一版中，我们从 Mike 的示例开始，展示了如何从底层构建，但是我们并不是真的从零开始的（我们借鉴了 Mike 的示例）。由于我们压根不会从零开始构建，所以在这一版中我要说出关于 D3 的一个真相：几乎所有的 D3 图表都是基于 Mike Bostock 的示例创建出来的。能把示例代码拿来根据你的需求进行修改，不仅是可视化工作中的基本技能，也是所有数据科学乃至编程中的基本技能。没有人什么都会，工作需要我们完成各种各样的任务。这一版就是要教会你做这样的事。

我们的模板复制了示例中的 CSS 样式代码：

```
{% extends "layout.html" %}
{% block body %}
<style>

.chart rect {
  fill: steelblue;
}

.chart text {
  fill: white;
  font: 10px sans-serif;
  text-anchor: middle;
}

</style>

  <div>
    <p class="lead">Total Flights by Month</p>
    <div id="chart"><svg class="chart"></svg></div>
  </div>
  <script src="/static/app.js"></script>
  <script>

  </script>
{% endblock %}
```

我们还把这个示例的 JavaScript 代码也放入了 *ch05/web/static/app.js*，并且稍作修改。*d3.tsv* 对我们没用，除非想把示例中的数据也拿过来。我们只需要稍微修改就能让示例工作起来。修改的行用粗体标出来了：

```
var width = 960,
    height = 350;

var y = d3.scale.linear()
    .range([height, 0]);
    // 等我们之后通过 d3.json 获取到数据就设置 domain
```


仅供非商业用途或交流学习使用

```
var chart = d3.select(".chart")
    .attr("width", width)
    .attr("height", height);

d3.json("/total_flights.json", function(data) {
  y.domain([0, d3.max(data, function(d) { return d.total_flights; })]);

  var barWidth = width / data.length;

  var bar = chart.selectAll("g") .data(data)
      .enter()
      .append("g")
      .attr("transform", function(d, i) {
          return "translate("+ i * barWidth + ",0)"; });

  bar.append("rect")
      .attr("y", function(d) { return y(d.total_flights); })
      .attr("height", function(d) { return height - y(d.total_flights); })
      .attr("width", barWidth - 1);

  bar.append("text")
      .attr("x", barWidth / 2)
      .attr("y", function(d) { return y(d.total_flights) + 3; })
      .attr("dy", ".75em")
      .text(function(d) { return d.total_flights; });
});
```

首先，使用 d3.json 方法把脚本指向我们的数据，指向服务器端的 */total_flights.json*。然后，我们只要把整个文件中所有 y 轴值的字段改为 total_flights 就可以了！这样我们得到了一个详细展示每个月航班总量变化的图表（见图 5-5）。

仅供非商业用途或交流学习使用

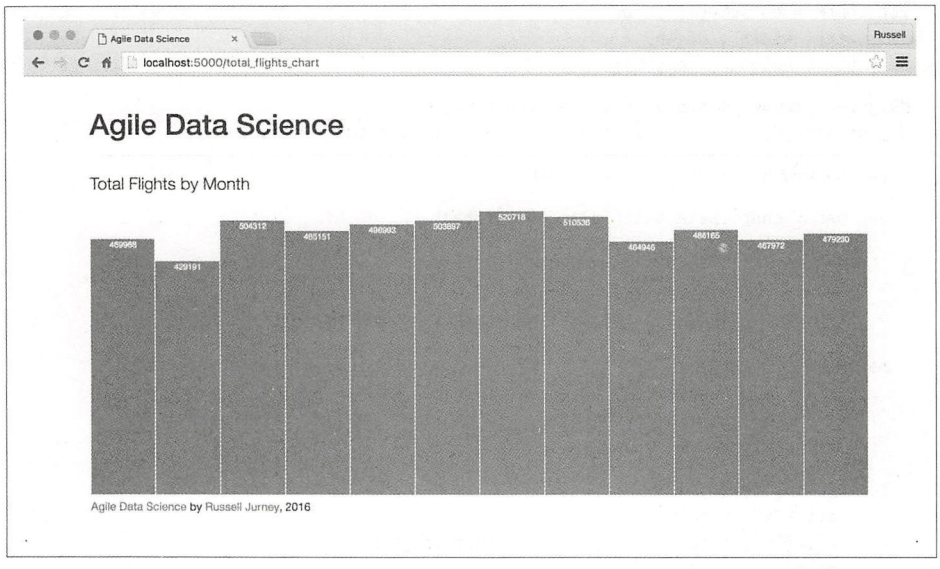


图 5-5　修改 Mike Bostock 的示例得到的图表

我们已经用 D3 创建了一个简单的图表。注意我们还没把图表弄得很精美，第一次总是不会特别完美的。我们先从简单实现数据可视化的图表开始，然后再为图表加上必要的装饰。

对第一张图表进行迭代

正如我们所介绍的一样，我们要对这张图表进行迭代，来帮助我们回答我们最初的问题："哪个月飞机出行最繁忙？"你能只看这张图表就回答这个问题吗？通过仔细观察，我发现是七月，不过我没法只凭一眼得出结论。

我们的问题实际上是在问：一年中，航班所属月份的众数是多少？维基百科把众数（*mode*）定义为"数据集中出现次数最多的值"。我们可以通过突出众数改进我们的图表，毕竟光从条形图来看结论并不明显。让我们修改 *app.js* 来把众数月高亮显示出来，让我们能一眼看出结果。我们需要创建 `varColor` 函数，当出现最大值时返回一个特殊的颜色。我们用 D3 的 `style`（*https://github.com/d3/d3-selection#selection_style*）方法把这个颜色应用到图表的长条上：

```
d3.json("/total_flights.json", function(data) {

    var defaultColor = 'steelblue';
    var modeColor = '#4CA9F5';

    var maxY = d3.max(data, function(d) { return d.total_flights; });
    y.domain([0, maxY]);
```

```
    var varColor = function(d, i) {
        if(d['total_flights'] == maxY) { return modeColor; }
        else { return defaultColor; }
    }
    var barWidth = width / data.length;
    var bar = chart.selectAll("g")
        .data(data)
        .enter()
        .append("g")
        .attr("transform", function(d, i) {
            return "translate("+ i * barWidth + ",0)"; });

    bar.append("rect")
        .attr("y", function(d) { return y(d.total_flights); })
        .attr("height", function(d) { return height - y(d.total_flights); })
        .attr("width", barWidth - 1)
        .style("fill", varColor);

    bar.append("text")
        .attr("x", barWidth / 2)
        .attr("y", function(d) { return y(d.total_flights) + 3; })
        .attr("dy", ".75em")
        .text(function(d) { return d.total_flights; });
});
```

根据结果，问题的答案已经显而易见了（见图 5-6）。

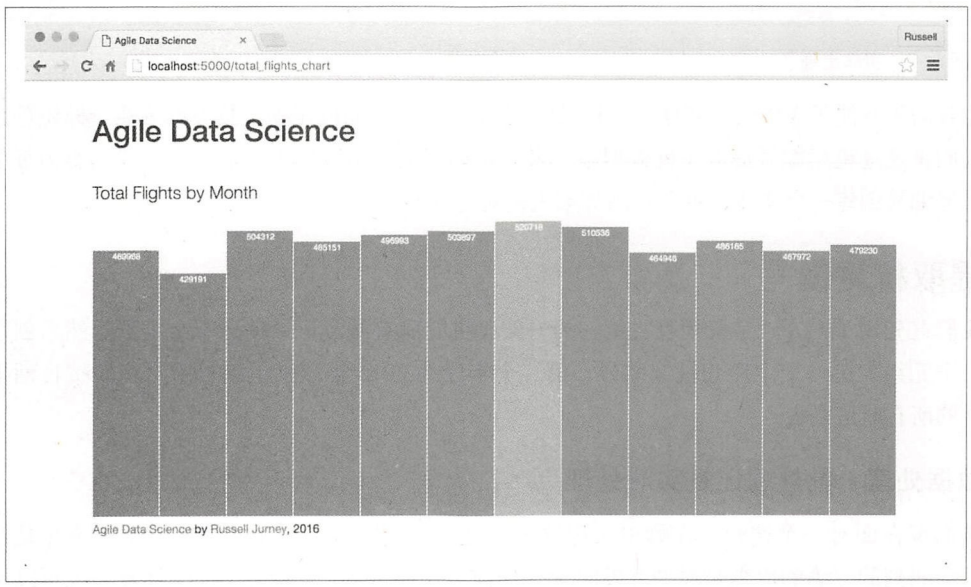


图 5-6　第二轮迭代后的图表

这样我们就讲完了数据价值金字塔第二层的表面要素。现在让我们探索另一个要素：实体提取。

提取“金属”（飞机（实体））

数据价值金字塔的图表层还有一个要素：实体提取。在创建图表时，我们把原子记录按属性分组做聚合，这其实是隐含的实体提取。实体一般是我们分析过程中的下一步。比如，前一章做了航班的列表和搜索，本章中对航班进行了聚合，这时我们会自然而然地想对航班的一些方面进行深挖：飞机、航空公司、机场等。图 5-7 展示了我们可能要提取的一些实体。

939 Flights

Carrier Origin Dest FlightDate TailNum N18223 FlightNum Submit

Airline	Flight Number	Origin	Destination	Date	Departure Time	Tail Number	Air Time	Distance
UA	1104	ANC	DEN	2015-01-01	1	N18223	268.0	2405.0
UA	1746	ATL	SFO	2015-01-01	741	N18223	300.0	2139.0
UA	1451	SFO	ORD	2015-01-01	1106	N18223	231.0	1846.0
UA	1623	ORD	ANC	2015-01-01	1844	N18223	407.0	2846.0
UA	1250	DEN	MCO	2015-01-02	807	N18223	181.0	1546.0
UA	1262	MCO	EWR	2015-01-02	1416	N18223	124.0	937.0
UA	1191	EWR	SFO	2015-01-02	1833	N18223	367.0	2565.0
UA	1204	SFO	IAH	2015-01-02	2328	N18223	210.0	1635.0

图 5-7　实体出现了！

让我们先专注于飞机这个实体。飞机是航班的“金属”，而机尾编号是飞机的唯一标识符。我们将通过机尾编号提取飞机数据集，来展示如何从原始数据中提取实体。我们要为每个机尾编号创建一个实体，可以列出那架飞机对应的全部航班。

提取机尾编号

我们首先用 `TailNum` 字段代表飞机，为一架飞机的所有航班记录创建一个索引。然后创建一个元组，第一个字段为机尾编号，第二个字段为 2015 年该机尾编号所对应的按日期排序的所有航班列表。

数据处理：批处理还是实时处理

我们现在面对一个选择：在哪里实现这个功能。在实际数据应用领域中，这种选择是经常会遇到的。总的原则是原型开发时选择任何一层都可以，可以用 Spark 计算，也可以用 HTML 模拟，但要尽可能地移到批处理阶段。

就本例而言，第一种选择是用 PySpark 按机尾编号对航班进行分组。这种方法把我们所有的处理都放到后端的批处理中，符合处理大量数据的要求。第二种方法是和之前一样使用 MongoDB 或者 Elasticsearch 查询航班记录索引，只是在网络应用程序的处理中稍作修改。

在本例中，我们选择用 PySpark 做航班分组，然后存储到 MongoDB 中。我们这么做是因为还要在其他分析过程中用这些数据来做表连接操作，尽管我们可以用 PySpark 直接从 Elasticsearch 里读取数据，但是把阶段性结果保存到可靠的批量存储中是很重要的，这真正实现了持久化和易于访问。我们知道用 Spark 可以轻松地随意伸缩操作，因此在这一层中实现数据处理是很保险的。

用 Spark 进行数据分组和排序

我们需要把数据按机尾编号进行分组来获取每架飞机对应的航班列表。航班由航空公司、日期、起飞地 / 目的地以及航班号进行区分。注意：对于飞机直接掉头飞回出发地的那种往返航班，通常我们会看到同一个航空公司同一个航班号在一天中出现了两次，一次去程，一次返程，所以我们加上了起降地信息。

参考 *ch05/extract_airplanes.py*。首先我们读取数据，然后筛选出我们需要的字段。丢掉一些用不到的字段可以帮助提升性能：

```
# 读取 Parquet 文件
on_time_dataframe = spark.read.parquet('data/on_time_performance.parquet')

# 筛选出我们需要用来区分航班的字段
flights = on_time_dataframe.rdd.map(lambda x:
 (x.Carrier, x.FlightDate, x.FlightNum, x.Origin, x.Dest, x.TailNum)
 )
```

现在，我们按机尾编号对航班记录进行分组，然后按照日期、航班号、起降地机场代码进行排序。注意：实现此目标的第一步是创建一个元组，元组的第一个字段为机尾编号，第二个字段为只包含一个元组的列表。只包含一个元组的列表有什么好处呢？ Python 中的列表可以像这样做加法：

```
a=[0]
b=[1]
c=a+b
print(c)
```

结果为：

```
[0, 1]
```

这里我们可以在归约（reduce）阶段把我们的列表加起来，因此我们需要在映射（map）阶段对列表进行初始化：

```
flights_per_airplane = flights\
 .map(lambda nameTuple: (nameTuple[5], [nameTuple[0:5]]))\
```

还要注意：我们在这个元组列表中把最后一个字段 TailNum 丢掉了。在同一个分组中，所有记录的 TailNum 都是一样的，并且作为键保存，因此这个字段是冗余的。这是一个风格问题；如果你喜欢，可以保留这个字段。

接下来，我们在归约阶段通过加法把键相同的值聚合为列表：

```
 .reduceByKey(lambda a, b: a + b)\
```

终于，我们获得了可以保存到 Mongo 中的字典类型变量。我们还依次按照日期、航班号、起飞地、目的地对航班列表进行了排序：

```
 .map(lambda tuple:
     {
       'TailNum': tuple[0],
       'Flights': sorted(tuple[1], key=lambda x: (x[1], x[2], x[3], x[4]))
     }
 )
```

看看我们得到了什么（可能需要执行几分钟）：

```
>db.flights_per_airplane.first()

{'Flights': [(u'AA', u'2015-01-01', 262, u'RSW', u'DFW'),
  (u'AA', u'2015-01-01', 2414, u'DFW', u'EWR'),
  (u'AA', u'2015-01-02', 1060, u'LAX', u'TPA'),
  (u'AA', u'2015-01-02', 1161, u'MIA', u'TPA'),
  (u'AA', u'2015-01-02', 1161, u'TPA', u'MIA'),
  (u'AA', u'2015-01-02', 1205, u'EWR', u'MIA'),
  (u'AA', u'2015-01-02', 1370, u'MIA', u'ORD'),
  (u'AA', u'2015-01-02', 2271, u'ORD', u'LAX'),
  (u'AA', u'2015-01-03', 346, u'ORD', u'LGA'),
  (u'AA', u'2015-01-03', 1192, u'LAX', u'ORD'),
  (u'AA', u'2015-01-03', 1209, u'TPA', u'LAX'),
  ...],
 'TailNum': u'N3MDAA'}
```

发布飞机到 Mongo 中

最后，我们把这些记录存入 MongoDB，这样我们可以通过机尾编号读取：

```
import pymongo_spark
pymongo_spark.activate()
flights_per_airplane.saveToMongoDB(
 'mongodb://localhost:27017/agile_data_science.flights_per_airplane'
 )
```

现在检查数据已经存入 MongoDB 中：

```
mongo agile_data_science

>db.flights_per_airplane.findOne()
```

```
{
        "_id" : ObjectId("5700092b8821240a5941fed2"),
        "TailNum" : "N249AU",
        "Flights" : [
                [
                        "US",
                        "2015-01-03",
                        837,
                        "STT",
                        "PHL"
                ],
    ...
        ]
}
```

用 Flask 提供飞机记录

我们可以看到如何通过机尾编号查询飞机，这是一种重要的访问模式，因为机尾编号是飞机的唯一标识符。这种数据是基础性的——它能让我们通过直接渲染预先计算好的数据来为页面添加功能。我们首先使用 *ch05/web/chart_flask.py* 中定义的控制器 */airplane/flights* 把这些航班以列表形式展示：

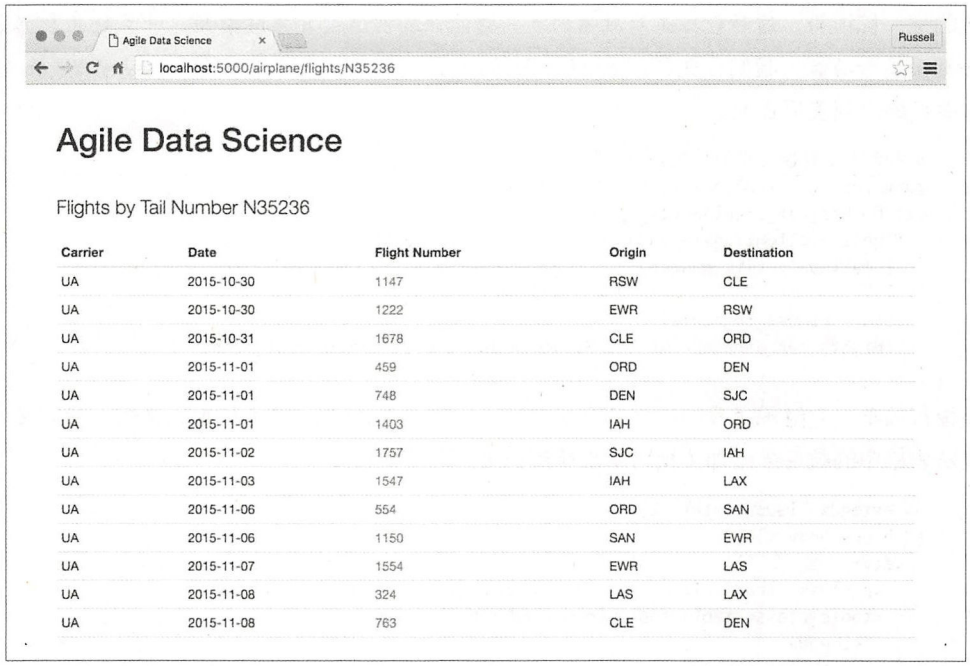
```
# 控制器：获取飞机的一条记录并展示
@app.route("/airplane/flights/<tail_number>")
def flights_per_airplane(tail_number):
  flights = client.agile_data_science.flights_per_airplane.find_one(
   {'TailNum': tail_number}
)
  return render_template(
   'flights_per_airplane.html', flights=flights, tail_number=tail_number
)
```

模板很简单。它继承了我们应用的全局布局，依赖了 Bootstrap 为表格提供样式。我们又一次给表格中的航班号添加了指向单次航班记录页面的链接：

```
{% extends "layout.html" %}
{% block body %}
  <div>
    <p class="lead">Flights by Tail Number {{tail_number}}</p>
    <table class="table table-condensed table-striped">
      <thead>
        <th>Carrier</th>
        <th>Date</th>
        <th>Flight Number</th>
        <th>Origin</th>
        <th>Destination</th>
      </thead>
      <tbody>
        {% for flight in flights['Flights'] %}
        <tr>
```

```
            <td>{{flight[0]}}</td>
            <td>{{flight[1]}}</td>
            <td>
              <a href="/on_time_performance?Carrier={{flight[0]}}
&FlightDate={{flight[1]}}
&FlightNum={{flight[2]}}">{{flight[2]}}</a>
            </td>
            <td>{{flight[3]}}</td>
            <td>{{flight[4]}}</td>
          </tr>
          {% endfor %}
        </tbody>
      </table>
    </div>
  {% endblock %}
```

这样，我们得到了一架飞机一年中执飞的所有航班的列表页面（见图 5-8）。



Carrier	Date	Flight Number	Origin	Destination
UA	2015-10-30	1147	RSW	CLE
UA	2015-10-30	1222	EWR	RSW
UA	2015-10-31	1678	CLE	ORD
UA	2015-11-01	459	ORD	DEN
UA	2015-11-01	748	DEN	SJC
UA	2015-11-01	1403	IAH	ORD
UA	2015-11-02	1757	SJC	IAH
UA	2015-11-03	1547	IAH	LAX
UA	2015-11-06	554	ORD	SAN
UA	2015-11-06	1150	SAN	EWR
UA	2015-11-07	1554	EWR	LAS
UA	2015-11-08	324	LAS	LAX
UA	2015-11-08	763	CLE	DEN

图 5-8　一个机尾编号对应的航班

使用索引保障数据库性能

然而，这个过程中有一个问题。我们的查询很慢！我们需要为 Mongo 添加索引来提升使用机尾编号查询对应航班记录的性能。如 *ch05/mongo.js* 所示。

现在，不妨让我们讨论一下索引。Mongo 中的索引和 MySQL 或者其他任何关系型数据库中的索引是类似的。它们使用 B 树来优化索引查询。当查询缓慢时，索引就要派上用场了。

首先，我们使用 explain 来确认查询不使用索引。输出显示，使用 BasicCursor（*https://docs.mongodb.com/v3.0/core/cursors/*）指针，获取一条返回记录共扫描了 13,533 个对象。

```
>db.flights_per_airplane.find({"TailNum": "N361VA"}).explain()
{
      "cursor" : "BasicCursor",
      "isMultiKey": false,
      "n":4,
      "nscannedObjects" : 13533,
      "nscanned": 13533,
      "nscannedObjectsAllPlans": 13533,
      "nscannedAllPlans": 13533,
      "scanAndOrder": false,
      "indexOnly": false,
      "nYields": 105,
      "nChunkSkips": 0,
      "millis": 28,
      "server": "Russells-MacBook-Pro-OLD-506.local:27017",
      "filterSet": false
}
```

然后，我们使用 ensureIndex 添加索引。创建索引很简单，我们只要选择要查询的字段，把它们作为 JSON 对象中的键，对应的值设为 1 即可：

```
>db.flights_per_airplane.ensureIndex({"TailNum": 1})
{
      "createdCollectionAutomatically": false,
      "numIndexesBefore": 1,
      "numIndexesAfter": 2,
      "ok":1
}
```

最后，我们再调用 explain，确保查询会使用索引：

```
>db.flights_per_airplane.find({"TailNum": "N361VA"}).explain()
{
      "cursor": "BtreeCursor TailNum_1",
      "isMultiKey": false,
      "n":4,
      "nscannedObjects": 4,
      "nscanned": 4,
      "nscannedObjectsAllPlans": 4,
      "nscannedAllPlans": 4,
      "scanAndOrder": false,
      "indexOnly": false,
```

```
        "nYields": 0,
        "nChunkSkips": 0,
        "millis": 3,
        "indexBounds": {
                "TailNum": [
                        [
                                "N361VA",
                                "N361VA"
                        ]
                ]
        },
        "server": "Russells-MacBook-Pro-OLD-506.local:27017",
        "filterSet": false
}
```

现在我们的查询会使用索引 `TailNum_1` 了，只要扫描四个对象。因此，查询可以立刻返回，我们的应用可以有不错的性能。

每当创建新的集合时，我们都要添加索引。如果忘记了，你可能有的时候能发现，有的时候不能发现。这是因为即使一条查询要进行全表扫描才能返回结果，如果用户不多的话，查询还是比较快的。然而当用户数量比较多的时候，这些问题就会暴露出来，所以最好在创建集合时就创建好索引。

添加回到新实体页面的链接

我们还要做一件事：为航班页面添加指向机尾编号页面的链接。我们需要编辑 *ch05/web/templates/flight.html* 和 *ch05/web/templates/search.html* 文件：

```
{% extends "layout.html" %}
{% block body %}
  <div>
    <p class="lead">Flight {{flight.FlightNum}}</p>
    <table class="table">
      <thead>
        <th>Airline</th>
        <th>Origin</th>
        <th>Destination</th>
        <th>Tail Number</th>
        <th>Date</th>
        <th>Air Time</th>
        <th>Distance</th>
      </thead>
      <tbody>
        <tr>
          <td>{{flight.Carrier}}</td>
          <td>{{flight.Origin}}</td>
          <td>{{flight.Dest}}</td>
          <td><a href="/airplane/flights/{{flight.TailNum}}">{{flight.TailNum}}
```

```
            </a>
          </td>
          <td>{{flight.FlightDate}}</td>
          <td>{{flight.AirTime}}</td>
          <td>{{flight.Distance}}</td>
        </tr>
      </tbody>
    </table>
  </div>
{% endblock %}
```

结果如图 5-9 所示。

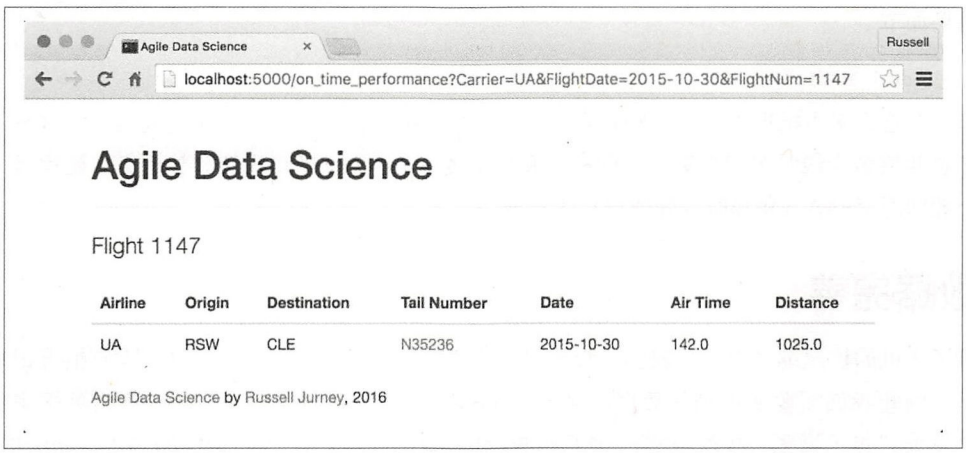


图 5-9　从航班页面回到飞机页面的链接

信息架构

现在我们的飞机页面已经上线了，我们有地方可以放在攀登数据价值金字塔的过程中创建出的任何有意思的数据表格、图表或者推荐。我们正在创建一种良好的信息架构。维基百科把信息架构定义为“共享信息环境中的结构化设计”。在我们构建共享信息环境（我们的网络应用以及团队内的深层存储）时，如果应用有逻辑结构，应用就更适合于浏览与共享。这对用户来说是有意义的，在他们提出问题并寻求答案的过程中，会反复研究我们创建出的各种实体。

评估飞机记录

既然我们在讨论飞机，那么让我们对这个刚刚创建的阶段性数据集进行评估。总共有多少架飞机？我们可以通过运行 *ch05/assess_airplanes.py* 获取总数：

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

```
# 读取 Parquet 文件
on_time_dataframe = spark.read.parquet('data/on_time_performance.parquet')
on_time_dataframe.registerTempTable("on_time_performance")

# 丢弃不需要的字段
tail_numbers = on_time_dataframe.map(lambda x: x.TailNum)
tail_numbers = tail_numbers.filter(lambda x: x != '')

# 用 distinct() 获取去重的机尾编号列表
unique_tail_numbers = tail_numbers.distinct()

# 使用 count() 统计去重的机尾编号总数
airplane_count = unique_tail_numbers.count()
print("Total airplanes: {}".format(airplane_count))
```

结果如下：

```
Total airplanes: 4897
```

哇，有这么多飞机！现在我开始好奇了：这些飞机都是什么型号的？制造商是谁？这些飞机总共值多少钱？要回答第一个问题，我们需要*完善*（*Enrich*）我们的数据集，使用另一个数据源：FAA（联邦航空管理局）注册表。

数据完善

有了飞机的机尾编号之后，我们还想要飞机的更多信息！这些信息不包含在准点情况记录中，因此我们需要使用别的数据源完善（enrich）我们的数据集。Techopedia 把数据完善定义为“表示增强、提炼或者改进原始数据的过程的术语”。我们所说的*完善*（*enrich*）指的是引入另一个数据集来增强已有的数据集，换句话说，就是表连接操作和*数据清洗*（*munging*）等附加处理。

网页表单逆向工程

我们需要的数据在 FAA 注册表里有。看看这个注册表里都有哪些好东西：制造商、型号、制造年份、机主，甚至引擎制造商和型号都有！这为我们的分析提供了很多可能。我们需要这个数据。

只是有一个问题：这些数据没有提供下载。这种情况很常见，也是我们说数据科学 90% 的工作是数据清洗的原因之一。为了获取这个数据集，我们需要爬取（scrape）数据，或者说每次从一个航空器注册编号[1]返回的页面中提取数据。我们可以使用 Python 轻松完成。

1 原文为 N-Number，根据 1944 年《芝加哥民航条约》规定，各国航空器都有统一的航空器注册编号，每个国家有不同的首码，注册在美国的航空器使用美国的首码 N，因而 FAA 注册表把美国的注册码称为 N-Number。——译者注

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

我们在示例代码中没有提供这些操作的脚本，因为我们不希望众多读者都去爬取 FAA 注册表（这可能会导致服务因为过载而崩溃）。你可以从书中复制粘贴或者抄代码来爬取别的页面，但是请不要爬取 FAA 注册表。

在开始编程之前，我们先审视查询表单的结果（见图 5-10）。我们对机尾编号 N933EV 的搜索 URL 如图 5-11 所示。你可以看到在查询 URL 中，航空器注册编号被编码为 *NNumbertxt* 参数，这表示结果表单可以通过 HTTP GET 请求获取。如果你不清楚网络的这些细节，参阅 RFC 2616 第 9 节的定义。不熟悉表单的人需要浏览第 3 节和第 4 节。另一种的表单类型是 POST 表单，我们后面会介绍到。爬取 GET 表单很容易。

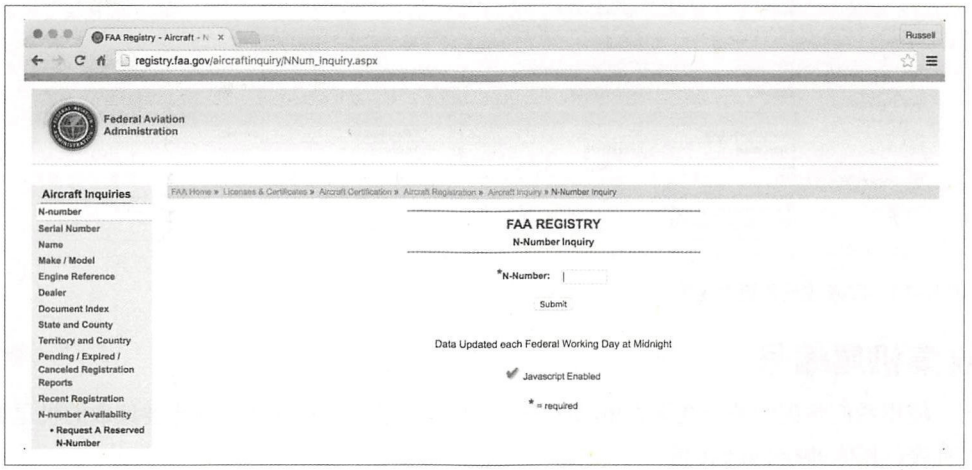


图 5-10　FAA 注册表航空器注册编号查询页面

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

图 5–11　FAA 注册表查询结果

收集机尾编号

为了使用我们学到的关于查询表单的知识，我们需要为我们的爬虫创建出要读取的机尾编号列表，用作 `NNumbertxt` 值。

我们使用 *ch05/save_tail_numbers.py* 实现：

```
# 读取 Parquet 文件
on_time_dataframe = spark.read.parquet('data/on_time_performance.parquet')
on_time_dataframe.registerTempTable("on_time_performance")

# 丢弃不需要的字段
tail_numbers = on_time_dataframe.map(lambda x: x.TailNum)
tail_numbers = tail_numbers.filter(lambda x: x != '')

# 用 distinct() 获取去重的机尾编号列表
unique_tail_numbers = tail_numbers.distinct()

# 通过 dataframe 存储为 JSON 对象。设置分区为 1 来输出到 1 个 JSON 文件中
unique_records = unique_tail_numbers.map(lambda x: {'TailNum': x})
unique_records.toDF().repartition(1).write.json("data/tail_numbers.json")
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

现在，在 bash 中运行：

```
$ ls data/tail_numbers.json/part*
data/tail_numbers.json/part-r-00000-1f29285f-55b2-4092-8c40-9d4b4c957f90
```

看一下：

```
$ head -5 data/tail_numbers.json/part*
```

结果如下：

```
{"TailNum":"N933EV"}
{"TailNum":"N917WN"}
{"TailNum":"N438WN"}
{"TailNum":"N3CHAA"}
{"TailNum":"N875AA"}
```

把文件名改得好记一些：

```
cp data/tail_numbers.json/part* data/tail_numbers.jsonl
```

现在可以开始爬数据了！

自动化表单提交

Python 的 `requests` 包（*http://docs.python-requests.org/en/master/*）很适合用来获取网页（用 Python 写爬虫的另一种方式是使用 Selenium（*http://www.seleniumhq.org/*），它可以自动化操作浏览器；我们后面会介绍到）。而 Python 的 `BeautifulSoup` 包（*https://www.crummy.com/software/BeautifulSoup/*）可以很方便地解析 HTML 来提取数据。

把这些工具串起来爬取数据很简单。让我们一步步介绍我们的爬虫脚本。

首先我们要使用第 3 章中用过的工具，读取用 PySpark 生成的 JSON 文件：

```
import sys, os, re
import time

sys.path.append("lib")
import utils

import requests
from bs4 import BeautifulSoup

tail_number_records = utils.read_json_lines_file('data/tail_numbers.jsonl')
```

接下来我们要遍历机尾编号，根据 */robots.txt* 规则（*http://www.robotstxt.org/guidelines.html*），记得在读取页面之前睡眠。如果没有先睡眠，你可能会不小心忘记调用睡眠而因为大量请求压垮一个网站：

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

```
aircraft_records = []
# 遍历机尾编号进行获取
for tail_number_record in tail_number_records:
  time.sleep(0.1) # 有必要再循环中先睡眠，否则会搞垮网站
  ...
```

在开发使用循环的脚本时，不要在循环中直接开发。把循环的第一个元素拿到 iPython 中，依次运行各操作一次。例如：

```
tail_number_record = tail_number_records[0]
```

然后粘贴循环中的操作，注意处理缩进。

接下来，我们使用机尾编号构建 URL，提交请求，解析返回的 HTML 页面：

```
# 使用机尾编号填写 URL 参数
BASE_URL =
  'http://registry.faa.gov/aircraftinquiry/NNum_Results.aspx?NNumbertxt={}'
tail_number = tail_number_record['TailNum']
url = BASE_URL.format(tail_number)

# 获取页面，解析 HTML
r = requests.get(url)
```

现在我们的表单可以自动化提交了！

从 HTML 中提取数据

下一步，我们需要解析请求返回的 HTML 页面，提取数据：

```
html = r.text
soup = BeautifulSoup(html)
```

现在我们需要检查网页（见图 5-12），使用 `BeautifulSoup` 找出相应的结构。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

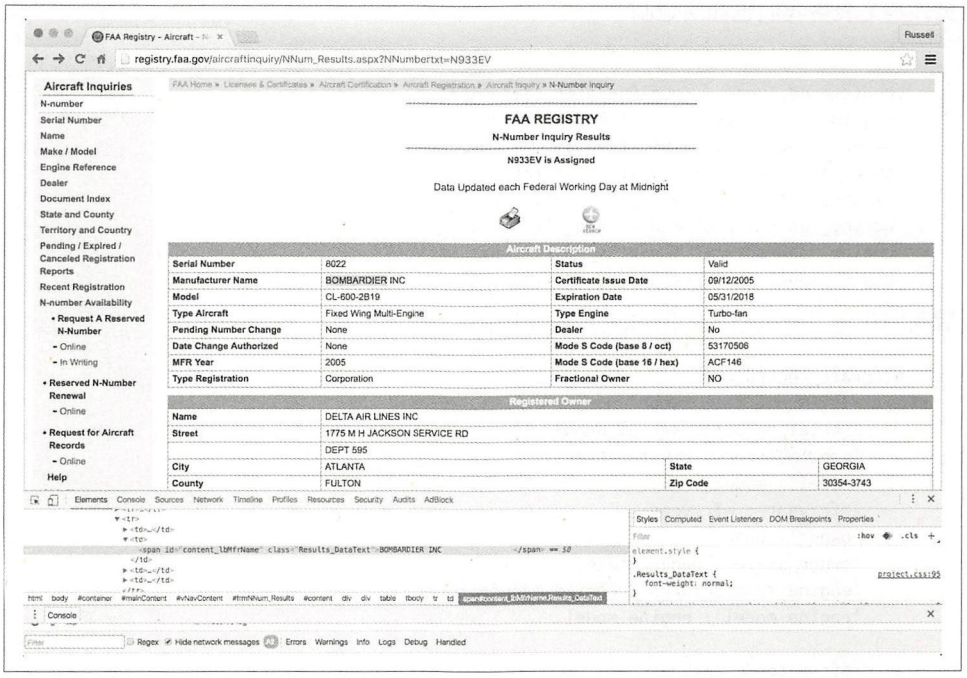


图 5-12　检查 FAA 注册表表格

这个页面高度依赖于 HTML 表格（*https://www.w3schools.com/html/html_tables.asp*），这样很好。我们感兴趣的数据在文档的表 5、表 6 和表 7 中。让我们看看表 5 飞行器描述的内容：

```
...
# 所有页面的表格结构都是一样的
try:
  aircraft_description = soup.find_all('table')[4]
  craft_tds = aircraft_description.find_all('td')
  serial_number = craft_tds[1].text.strip()
  manufacturer = craft_tds[5].text.strip()
  model = craft_tds[9].text.strip()
  mfr_year = craft_tds[25].text.strip()
```

使用 `BeautifulSoup` 的 `find_all` 我们可以得到页面上所有表格的列表，并选取表 5（下标为 4，从 0 开始）。下一步，我们把这张表中 `td` 元素的列表放到 `craft_tds` 中，然后输出来看字段结构。注意：我们调用了 `text` 方法获取了 `td` 元素的文本，然后调用 `str.strip` 处理这个值。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

对另外两个表格我们也如法炮制：

```
...
registered_owner = soup.find_all('table')[5]
reg_tds = registered_owner.find_all('td')
owner = reg_tds[1].text.strip()
owner_state = reg_tds[9].text.strip()
airworthiness = soup.find_all('table')[6]
worthy_tds = airworthiness.find_all('td')
engine_manufacturer = worthy_tds[1].text.strip()
engine_model = worthy_tds[5].text.strip()
```

最后我们获得了一条记录，把它添加到列表中：

```
aircraft_record = {
    'TailNum': tail_number,
    'serial_number': serial_number,
    'manufacturer': manufacturer,
    'model': model,
    'mfr_year': mfr_year,
    'owner': owner,
    'owner_state': owner_state,
    'engine_manufacturer': engine_manufacturer,
    'engine_model': engine_model,
  }
aircraft_records.append(
  aircraft_record
)
print(aircraft_record)
```

无法获取记录是这个脚本中唯一要处理的其他情况。在脚本执行了几轮之后，你会看到脚本出现异常挂掉了。这在爬取数据时经常发生——这是爬取数据过程的一部分；永远不会100% 成功的。检查了脚本挂掉的原因之后，根据问题，你很可能可以直接获取异常，打印一个错误，接受这种操作中一部分数据的丢失：

```
...
except IndexError, e:
  print("Missing {} record: {}".format(tail_number, e))
```

注意，你可以每条记录以 JSON 格式打印一行，也可以像我们这里所做的这样最后一起输出：

```
utils.write_json_lines_file(
  aircraft_records, 'data/faa_tail_number_inquiry.jsonl'
)
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

评价完善后的数据

现在我们已经把机尾编号数据存储在了 *data/faa_tail_number_inquiry.jsonl* 中，让我们来看一看。首先我们想知道我们成功获取了多少条记录，包括绝对值和百分比。

在 bash 中，运行：

```
head -5 data/faa_tail_number_inquiry.jsonl
```

结果如下：

```
{
  "engine_model": "CF34 SERIES",
  "engine_manufacturer": "GE",
  "owner_state": "GEORGIA",
  "serial_number": "8022",
  "owner": "DELTA AIR LINES INC",
  "TailNum": "N933EV",
  "model": "CL-600-2B19",
  "mfr_year": "2005",
  "manufacturer": "BOMBARDIER INC"
}
{
  "engine_model": "CFM56-7B24",
  "engine_manufacturer": "CFM INTL",
  "owner_state": "TEXAS",
  "serial_number": "36624",
  "owner": "SOUTHWEST AIRLINES CO",
  "TailNum": "N917WN",
  "model": "737-7H4",
  "mfr_year": "2008",
  "manufacturer": "BOEING"
}
...
```

我们可以通过 wc 统计记录总数，具体命令为 `wc -l data/faa_tail_number_inquiry.jsonl`：

```
4272 data/faa_tail_number_inquiry.jsonl
```

现在，如果数据集很大，让我们回到 PySpark 中读取数据来统计记录总数：

```
# 读取 FAA 航空器注册编号查询记录
faa_tail_number_inquiry = spark.read.json('data/faa_tail_number_inquiry.jsonl')
faa_tail_number_inquiry.show()

# 统计记录总数
faa_tail_number_inquiry.count()
```

结果为：4272。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

要看如何共同使用这两个数据集，让我们把这个数据集与去重的机尾编号表做表连接，看看有多少条命中：

```
# 读取去重的机尾编号表
unique_tail_numbers = spark.read.json('data/tail_numbers.jsonl')
unique_tail_numbers.show()

# 左外连接机尾编号表与查询记录表，查看有多少能连接起来
tail_num_plus_inquiry = unique_tail_numbers.join(
  faa_tail_number_inquiry,
  unique_tail_numbers.TailNum == faa_tail_number_inquiry.TailNum,
  'left_outer'
)
tail_num_plus_inquiry.show()

# 计算记录总数与成功连接的记录数
total_records = tail_num_plus_inquiry.count()
join_hits = tail_num_plus_inquiry.filter(
  tail_num_plus_inquiry.owner.isNotNull()
).count()

# 这里是纯 Python，我们可以计算并输出百分比 ...
hit_ratio = float(join_hits)/float(total_records)
hit_pct = hit_ratio * 100
print("Successful joins: {:.2f}%".format(hit_pct))
```

结果为 `Successful joins: 83.65%`。继续这个例子，接下来我们可能要查询没能成功连接的记录结构，看看是不是随机且可忽略的，或者是不是某个字段的值是同一类的，这样我们就要注意后续分析中使用表连接的结果会导致一些偏差的情况。

现在我们已经往我们的数据集中添加了一些有趣的新数据，下一章我们就会深入探索它们。

本章小结

在本章中，我们学习了通过表格和图表梳理数据的结构。我们也已经开始引入外部数据集来使我们的数据更加丰富，为我们提供新的分析维度。通过这样做，我们比上一章更深入地梳理了数据资产。在我们继续攀登数据价值金字塔的时候，这些技术会时常用到。

现在让我们进入数据价值栈的下一阶段——报表。

第 6 章

通过报表探索数据

接下来就是我们的第三个敏捷开发冲刺周期了，我们要把图表页面进一步开发为完整的报表。在这一步骤中，我们会给图表增添交互性，把静态页面变成动态页面，通过链接的网络、表和图表的相关实体使得数据可供探索。这些都是数据价值金字塔中报表阶段的特性（见图 6-1）。

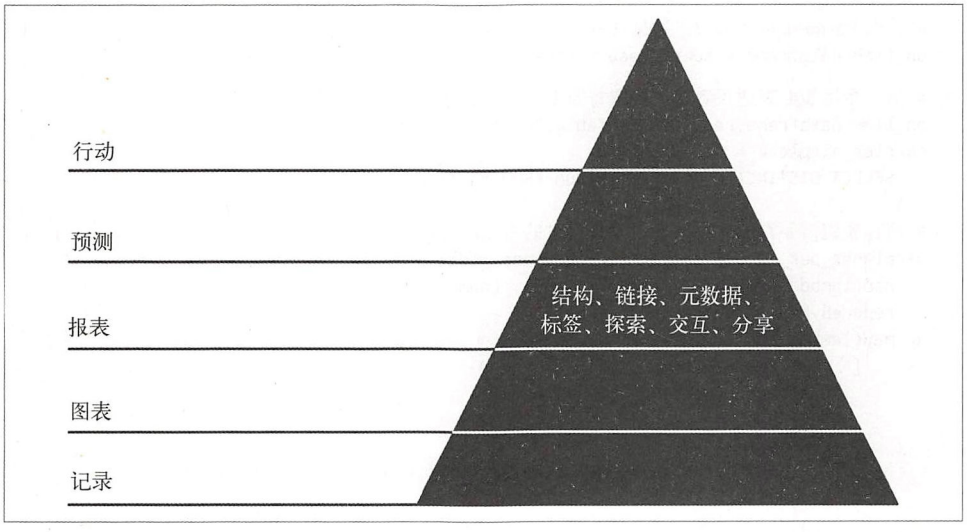


图 6-1　第三层：通过报表进行探索

本章的示例代码在 *Agile_Data_Code_2/tree/master/ch06* 中。克隆代码仓库，跟上我们的脚步！

```
git clone https://github.com/rjurney/Agile_Data_Code_2.git
```

提取航空公司为实体

为了构建报表，我们需要组合数据集的各种视图。构建这些视图对应的工作是枚举实体。我们在上一章中创建了实体“飞机”，它会是我们继续创建其他实体及建立实体间的关联关系以构建报表的基础。如上一章所述，在我们开始创建数据的视图之前，我们需要构建一个网页来展示图表与表格。那么下面我们就来创建一个新的实体“航空公司”，并且为每家航空公司提供一个页面。

我们从收集一家特定航空公司所有飞机的机尾编号入手。每个商业航班都是由某一家航空公司运营的，各个航空公司又拥有各自的机队，以及基地机场设施和人员，这些都是航空公司业务的核心资产。我们已经为每架飞机创建了页面，因此我们可以利用这一数据来创建每个航空公司的全部机尾编号列表。

使用 PySpark 把航空公司定义为飞机的分组

我们从准备每个航空公司的飞机机尾编号列表入手，代码如 *ch06/ extract_airlines.py* 文件所展示的。这会成为航空公司页面的基础：

```
# 读取 Parquet 格式的整点数据文件
on_time_dataframe = spark.read.parquet('data/on_time_performance.parquet')

# 第一步用 SQL 表达很容易：提取每家航空公司去重的机尾编号列表
on_time_dataframe.registerTempTable("on_time_performance")
carrier_airplane = spark.sql(
  "SELECT DISTINCT Carrier, TailNum FROM on_time_performance"
  )
# 现在我们需要存储每家航空公司排序过的机尾编号列表，以及机队飞机总数
airplanes_per_carrier = carrier_airplane.rdd\
  .map(lambda nameTuple: (nameTuple[0], [nameTuple[1]]))\
  .reduceByKey(lambda a, b: a + b)\
  .map(lambda tuple:
      {
        'Carrier': tuple[0],
        'TailNumbers': sorted(
          filter(
           lambda x: x != '', tuple[1] # 处理机尾编号为空字符串的情况
           )
          ),
        'FleetCount': len(tuple[1])
      }
    )
```

```
airplanes_per_carrier.count() # 14

# 存入 Mongo 表 airplanes_per_carrier 中
import pymongo_spark
pymongo_spark.activate()
airplanes_per_carrier.saveToMongoDB(
  'mongodb://localhost:27017/agile_data_science.airplanes_per_carrier'
)
```

在 MongoDB 中查询航空公司数据

下面来验证数据已经在 Mongo 中：`db.airplanes_per_carrier.find()`，返回如下：

```
{"_id": ..., "TailNumbers": ["N502NK", ...], "Carrier": "NK", "FleetCount": 79 }
{"_id": ..., "TailNumbers": ["N0EGMQ", ...], "Carrier": "MQ", "FleetCount": 204 }
{"_id": ..., "TailNumbers": ["N281VA", ...], "Carrier": "VX", "FleetCount": 57 }
```

在 Flask 中构建航空公司页面

下面我们要为航空公司页面创建控制器。我们的 Flask 控制器很简单，它接收描述航空公司的相关代码，返回一个包含飞机列表的页面，飞机以机尾编号的形式标示，列表从 Mongo 中读取：

```
@app.route("/airline/<carrier_code>")
def airline(carrier_code):
  airline_airplanes = client.agile_data_science.airplanes_per_carrier.find_one(
    {'Carrier': carrier_code}
  )
  return render_template(
    'airlines.html',
    airline_airplanes=airline_airplanes,
    carrier_code=carrier_code
  )
```

我们的模板代码则为每个机尾编号创建了对应的 HTML 元素，如 *ch06/web/templates/airlines.html* 所示：

```
{% extends "layout.html" %}
{% block body %}
  <p class="lead">Airline {{carrier_code}}</p>
  <h4>Fleet: {{airline_airplanes.FleetCount}} Planes</h4>
  <ul class="nav nav-pills">
    {% for tail_number in airline_airplanes.TailNumbers -%}
    <li class="button">
      <a href="/airplane/{{tail_number}}">{{tail_number}}</a>
    </li>
    {% endfor -%}
  </ul>
{% endblock %}
```

我们得到了一个简易的航空公司页面，列出了航空公司的机队（见图 6-2）。不用担心，我们稍后还会再装点这个页面的。在交付好产品之前，交付一些难看的东西也是很有必要的！

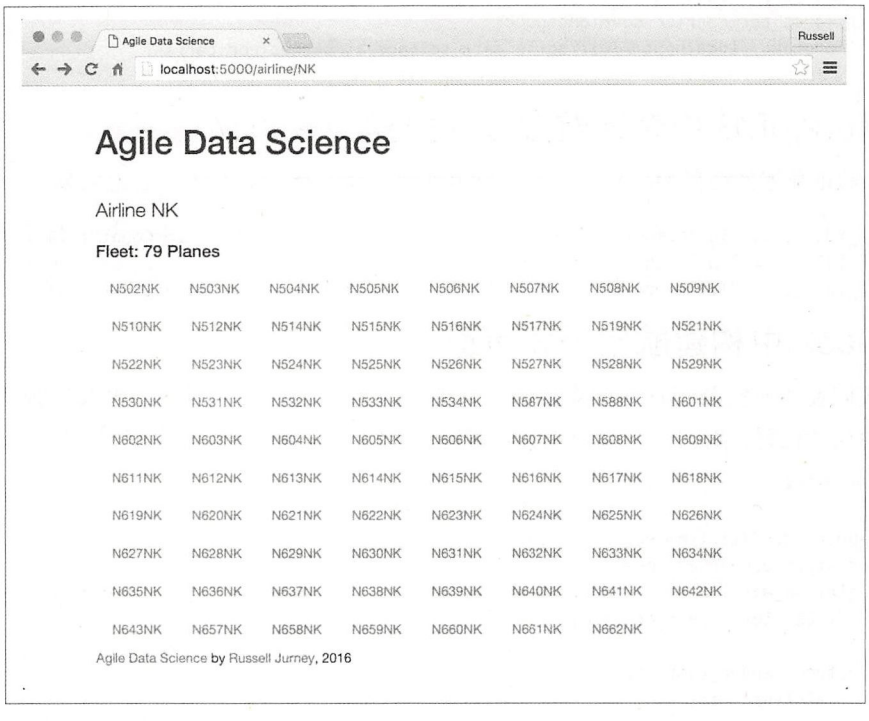


图 6-2　航空公司页面

添加回到航空公司页面的链接

创建了航空公司这个实体的页面之后，我们要为先前创建的飞机页面添加跳转回航空公司页面的链接，还有第 5 章中创建的搜索页面和航班页面也需要添加。我们可以修改飞机、航班及搜索页面的模板来实现这个功能。

我们要在飞机页面上添加跳转链接，如 *ch06/web/templates/flights_per_airplane.html* 所示：

```
<tbody>
  {% for flight in flights['Flights'] %}
  <tr>
    <td><a href="/airline/{{flight[0]}}">{{flight[0]}}</a></td>
    <td>{{flight[1]}}</td>
    <td><a href="/on_time_performance?Carrier={{flight[0]}}&FlightDate=
      {{flight[1]}}&FlightNum={{flight[2]}}">{{flight[2]}}</a></td>
    <td>{{flight[3]}}</td>
```

```
    <td>{{flight[4]}}</td>
  </tr>
  {% endfor %}
</tbody>
```

而在 *ch06/web/templates/flight.html* 中：

```
<tbody>
  <tr>
    <td><a href="/airline/{{flight.Carrier}}">{{flight.Carrier}}</a></td>
    <td>{{flight.Origin}}</td>
    <td>{{flight.Dest}}</td>
    <td><a href="/airplane/flights/{{flight.TailNum}}">
      {{flight.TailNum}}</a></td>
    <td>{{flight.FlightDate}}</td>
    <td>{{flight.AirTime}}</td>
    <td>{{flight.Distance}}</td>
  </tr>
</tbody>
```

还需修改 *ch06/web/templates/search.html*：

```
{% for flight in flights %}
  <tr>
    <td><a href="/airline/{{flight.Carrier}}">{{flight.Carrier}}</a></td>
    <td><a href="/on_time_performance?Carrier={{flight.Carrier}}&FlightDate=
      {{flight.FlightDate}}&FlightNum={{flight.FlightNum}}">{{flight.FlightNum}}
    </a></td>
    <td>{{flight.Origin}}</td>
    <td>{{flight.Dest}}</td>
   <td>{{flight.FlightDate}}</td>
   <td>{{flight.DepTime}}</td>
   <td><a href="/airplane/{{flight.TailNum}}">{{flight.TailNum}}</a></td>
   <td>{{flight.AirTime}}</td>
   <td>{{flight.Distance}}</td>
  </tr>
{% endfor %}
```

创建一个包括所有航空公司的主页

但是谁记得住那些航空公司代码呢？（好吧，我能记住）我们需要给用户一个入口，所以我们要创建一个主页，列出在美国运营的所有航空公司。

控制器部分很简单，只有六行代码。我们可以复用 MongoDB 表 `air lines_per_carrier`，这一次可以不考虑机尾编号，只使用 find 查询航空公司代码。我们同时把这个页面设置为本应用的默认首页 *index.html*：

```
@app.route("/")
@app.route("/airlines")
@app.route("/airlines/")
```

```
def airlines():
  airlines = client.agile_data_science.airplanes_per_carrier.find()
  return render_template('all_airlines.html', airlines=airlines)
```

模板部分则与一个航空公司的页面模板类似：

```
{% extends "layout.html" %}
{% block body %}
  <!-- Navigation guide -->
  / <a href="/airlines">Airlines</a>

  <p class="lead">US Domestic Airlines</p>
  <ul class="nav nav-pills">
    {% for airline in airlines -%}
    <li class="button">
      <a href="/airline/{{airline.Carrier}}">{{airline.Carrier}}</a>
    </li>
    {% endfor -%}
  </ul>
{% endblock %}
```

这样我们就得到了一个让用户可以简单而高效地浏览航空世界的方法（见图 6-3）。

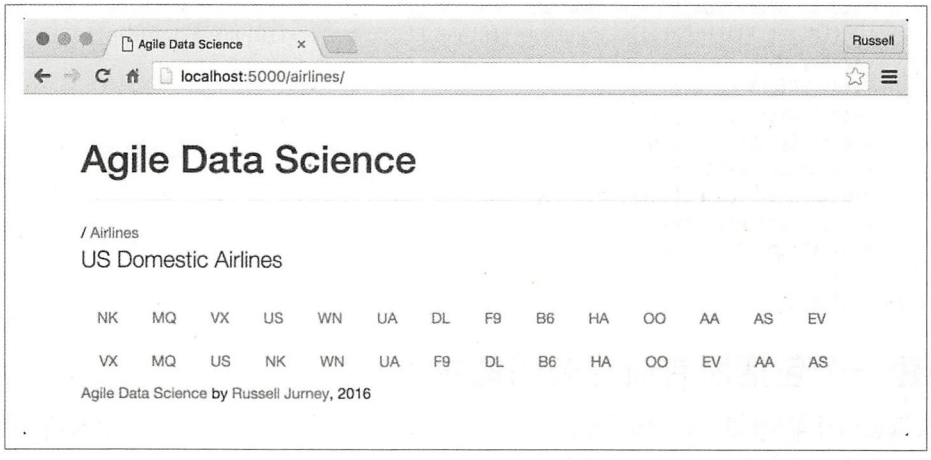


图 6-3　航线主页

整理半结构化数据的本体关系

现在我们可以尽情探索航空公司、飞机、航班数据了！这是一件大事，对吧？可能算不上，但这是一个好的开始。让我们进一步扩展这个功能，给航班页面里面的飞机和航空公司加上链接。

现在我们可以在查看航班时看到飞机和航空公司的信息——它们的属性及其之间的关系（见图 6-4）。这样的结构非常清晰，并且是推荐的一种简单的形式。

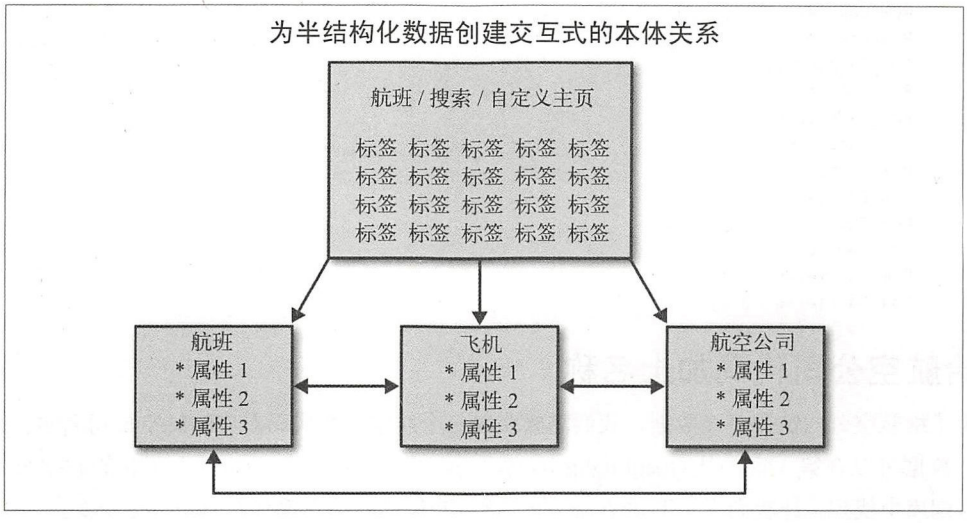


图 6-4　页面结构

我们正在做的事情可以被描述为创建半结构化数据间交互式的本体关系。把我们的流程围绕着构建这一结构关系拆开，对我们来说有几个作用。首先，它创建了一小批工作任务（每个实体一个），可以有效地进入敏捷开发的冲刺。这实现了一种数据敏捷性，并可以将我们的应用程序扩展为更适合浏览的状态。而这反过来又让用户可以自由单击并探索数据集，从而将团队与实际数据的情况连接起来。正如你现在所知，这是敏捷数据科学的一个主题。

改进航空公司页面

我们有了航空公司的页面，下面就给它加上一些多媒体内容，包括文本和图片。我们首先从主数据集中获取航空公司代码列表：

```
# 读取 Parquet 格式的整点数据文件
on_time_dataframe = spark.read.parquet('data/on_time_performance.parquet')

# 第一步用 SQL 表达很容易：提取每家航空公司去重的
# 机尾编号列表
on_time_dataframe.registerTempTable("on_time_performance")
carrier_codes = spark.sql(
  "SELECT DISTINCT Carrier FROM on_time_performance"
  )
carrier_codes.collect()
```

这样我们就得到了准点情况数据中出现的所有航空公司代码列表：

```
[Row(Carrier=u'AA'),
 Row(Carrier=u'NK'),
 Row(Carrier=u'HA'),
 Row(Carrier=u'AS'),
 Row(Carrier=u'B6'),
 Row(Carrier=u'UA'),
 Row(Carrier=u'US'),
 Row(Carrier=u'OO'),
 Row(Carrier=u'VX'),
 Row(Carrier=u'WN'),
 Row(Carrier=u'DL'),
 Row(Carrier=u'EV'),
 Row(Carrier=u'F9'),
 Row(Carrier=u'MQ')]
```

给航空公司代码加上名称

为了给航空公司链接更多数据，我们需要获取每个航空公司代码对应的航空公司名字。这一数据可以在第 5 章中从 OpenFlights（*http://openflights.org/data.html*）上下载的航空公司数据库中找到。让我们看一下 *airlines.dat*，这里我们已经把它重命名为 *airlines.csv*：

```
cat data/airlines.csv | grep '"DL"\|"NW"\|"AA"'
```

这里列出了一部分航空公司：

```
24,"American Airlines",\N,"AA","AAL","AMERICAN","United States","Y"
2009,"Delta Air Lines",\N,"DL","DAL","DELTA","United States","Y"
3731,"Northwest Airlines",\N,"NW","NWA","NORTHWEST","United States","Y"
```

OpenFlights 数据表中的字段依次为航空公司 ID、航空公司名字、别称、两字母 IATA 代码、三字母 ICAO 代码、呼号、国家、是否活跃。让我们在 PySpark 中读取并查看这份数据。代码如 *ch06/add_name_to_airlines.py* 所示：

```
airlines = spark.read.format('com.databricks.spark.csv')\
  .options(header='false', nullValue='\N')\
  .load('data/airlines.csv')
airlines.show()
```

结果如下所示：

```
+---+--------------------+----+---+---+---------------+---------------+---+
| C0|                  C1|  C2| C3| C4|             C5|             C6| C7|
+---+--------------------+----+---+---+---------------+---------------+---+
|  1|      Private flight|null|  -|N/A|               |               |  Y|
|  2|         135 Airways|null|   |GNL|        GENERAL| United States|  N|
|  3|       1Time Airline|null| 1T|RNX|        NEXTIME|  South Africa|  Y|
|  4|2 Sqn No 1 Elemen...|null|   |WYT|               |United Kingdom|  N|
...
```

还有：

```
# 有没有达美（Delta）航空？
airlines.filter(airlines.C3 == 'DL').show()
```

结果如下：

```
+----+---------------+----+---+---+-----+-------------+---+
|  C0|             C1|  C2| C3| C4|   C5|           C6| C7|
+----+---------------+----+---+---+-----+-------------+---+
|2009|Delta Air Lines|null| DL|DAL|DELTA|United States|  Y|
+----+---------------+----+---+---+-----+-------------+---+
```

我们仅保留数据中的航空公司名字和两个字母的航空公司代码，然后把它和准点情况数据集中去重的航空公司代码做表连接：

```
# 丢掉除名字字段 C1 和航空公司代码字段 C2 以外的字段
airlines.registerTempTable("airlines")
airlines = spark.sql("SELECT C1 AS Name, C3 AS CarrierCode from airlines")

# 把前面得到的 14 家航空公司代码与 airlines 表做表连接
# 以获取我们需要的航空公司数据集
our_airlines = carrier_codes.join(
  airlines, carrier_codes.Carrier == airlines.CarrierCode
)
our_airlines = our_airlines.select('Name', 'CarrierCode')
our_airlines.show()
```

结果如下：

```
+--------------------+-----------+
|                Name|CarrierCode|
+--------------------+-----------+
|   American Airlines|         AA|
|     Spirit Airlines|         NK|
|   Hawaiian Airlines|         HA|
|     Alaska Airlines|         AS|
|     JetBlue Airways|         B6|
|     United Airlines|         UA|
|          US Airways|         US|
|             SkyWest|         OO|
|      Virgin America|         VX|
|  Southwest Airlines|         WN|
|     Delta Air Lines|         DL|
|Atlantic Southeas...|         EV|
|   Frontier Airlines|         F9|
|American Eagle Ai...|         MQ|
+--------------------+-----------+
```

最后，把这一阶段性数据以 JSON 格式存储：

```
our_airlines.repartition(1).write.json("data/our_airlines.json")
```

接下来，还是复制到一个 JSON 行文件中：

```
cp data/our_airlines.json/part* data/our_airlines.jsonl
```

然后，我们就可以使用 cat data/our_airlines.jsonl 粗略地看一看数据了：

```
{"Name":"American Airlines","CarrierCode":"AA"}
{"Name":"Spirit Airlines","CarrierCode":"NK"}
{"Name":"Hawaiian Airlines","CarrierCode":"HA"}
{"Name":"Alaska Airlines","CarrierCode":"AS"}
{"Name":"JetBlue Airways","CarrierCode":"B6"}
{"Name":"United Airlines","CarrierCode":"UA"}
{"Name":"US Airways","CarrierCode":"US"}
{"Name":"SkyWest","CarrierCode":"OO"}
{"Name":"Virgin America","CarrierCode":"VX"}
{"Name":"Southwest Airlines","CarrierCode":"WN"}
{"Name":"Delta Air Lines","CarrierCode":"DL"}
{"Name":"Atlantic Southeast Airlines","CarrierCode":"EV"}
{"Name":"Frontier Airlines","CarrierCode":"F9"}
{"Name":"American Eagle Airlines","CarrierCode":"MQ"}
```

整合维基百科内容

现在我们获取了航空公司的名字，我们可以通过维基百科获取各家航空公司的各种信息，比如简介、标志、公司官网！要这么做的话，我们要使用 Python 的 wikipedia（*https://pypi.python.org/pypi/wikipedia/*）包，它封装了 MediaWiki API（*https://www.mediawiki.org/wiki/API:Main_page*）。我们要再次使用 BeautifulSoup 来解析网页的 HTML 代码。

如 *ch06/enrich_airlines_wikipedia.py* 所示：

```
import sys, os, re
sys.path.append("lib")
import utils

import wikipedia
from bs4 import BeautifulSoup
import tldextract

# 读取航空公司信息
our_airlines = utils.read_json_lines_file('data/our_airlines.jsonl')

# 创建一个新的包含维基百科数据的列表
with_url = []
for airline in our_airlines:
  # 获取航空公司名字对应的维基百科页面
  wikipage = wikipedia.page(airline['Name'])

  # 获取简介
  summary = wikipage.summary
  airline['summary'] = summary

  # 获取页面的 HTML 源码
  page = BeautifulSoup(wikipage.html())

# 任务：从右侧 'vcard' 栏获取标志
```

```
    # 1) 获取 vcard 表
    vcard_table = page.find_all('table', class_='vcard')[0]
    # 2) 标志总是表内的第一张图片
    first_image = vcard_table.find_all('img')[0]
    # 3) 保存图片 URL
    logo_url = 'http:' + first_image.get('src')
    airline['logo_url'] = logo_url

    # 任务：获取公司的官网
    # 1) 找到 'Website' 表头单元格
    th = page.find_all('th', text='Website')[0]
    # 2) 获取父 tr 元素
    tr = th.parent
    # 3) 找到该 tr 元素中的链接标签 a
    a = tr.find_all('a')[0]
    # 4) 最后，获取 a 标签的 href 属性
    url = a.get('href')
    airline['url'] = url

    # 获取 URL 中的域名
    url_parts = tldextract.extract(url)
    airline['domain'] = url_parts.domain + '.' + url_parts.suffix

    with_url.append(airline)

utils.write_json_lines_file(with_url, 'data/our_airlines_with_wiki.jsonl')
```

把扩充过的航空公司表发布到 MongoDB

我们在本节中还没有用到 Mongo，原始数据集经过了两轮扩充，中间数据集都没有存入 Mongo 中。没关系！在敏捷数据科学中，我们用数据库来发布数据，而不总是把它维持在中间状态。

然而，现在我们要把扩充过的航空公司表放到先前创建的航空公司网页上。为了实现这个目标，我们需要把新的数据存到 Mongo 中。由于我们已经有了准备好的 JSON 文件，因此可以使用 `mongoimport` 命令直接把它读取到 Mongo 中：

```
mongoimport -d agile_data_science -c airlines \
        --file data/our_airlines_with_wiki.jsonl
```

验证数据：

```
$ mongo agile_data_science
>db.airlines.findOne();
{
        "_id": ObjectId("57c0e656818573ed12d584d1"),
        "CarrierCode": "AA",
        "url": "http://www.aa.com",
        "logo_url": "http://upload.wikimedia.org/.../300px-American...
                    _2013.svg.png",
        "Name": "American Airlines",
        "summary": "American Airlines, Inc. (AA), commonly referred to as
                    American..."
}
```

在网页上扩充航空公司信息

我们已经扩充了 Mongo 中航空公司的记录，现在要修改 Flask 控制器为 */airline* 页面增加这些数据。如 *ch06/web/report_flask.py* 所示：

```
# 控制器：获取飞机实体页面
@app.route("/airline/<carrier_code>")
def airline(carrier_code):
  airline_summary = client.agile_data_science.airlines.find_one(
  {'CarrierCode': carrier_code}
)
airline_airplanes = client.agile_data_science.airplanes_per_carrier.find_one(
  {'Carrier': carrier_code}
)
return render_template(
  'airlines.html',
  airline_summary=airline_summary,
  airline_airplanes=airline_airplanes,
  carrier_code=carrier_code
)
```

下面我们修改模板 *ch06/web/templates/airlines.html*，以包含维基百科数据：

```
{% extends "layout.html" %}
{% block body %}
  <!-- 导航栏 -->
  / <a href="/airlines">Airlines</a>
    / <a href="/airline/{{carrier_code}}">{{carrier_code}}</a>

  <!- 标志 -->
  <img src="{{airline_summary.logo_url}}"style="float: right;"/>

<p class="lead">
  <!-- 航空公司名字和网站 -->
```

```
    {{airline_summary.Name}}
    / <a href="{{airline_summary.url}}">{{airline_summary.domain}}</a>
  </p>

  <!-- 简介 -->
  <p style="text-align: justify;">{{airline_summary.summary}}</p>
  <h4>Fleet: {{airline_airplanes.FleetCount}} Planes</h4>
  <ul class="nav nav-pills">
    {% for tail_number in airline_airplanes.TailNumbers -%}
    <li class="button">
      <a href="/airplane/{{tail_number}}">{{tail_number}}</a>
    </li>
    {% endfor -%}
  </ul>
{% endblock %}
```

经过一番优化，我们得到了一个大大改善的航空公司页面（见图 6-5）。这一修改意义何在？虽然你的数据可能不会这么轻易地使用维基百科这样的公共数据集进行扩充，但此示例展示了如何合并来自不同来源的数据（不管是私有数据还是公共数据）来组成更好的实体页面。

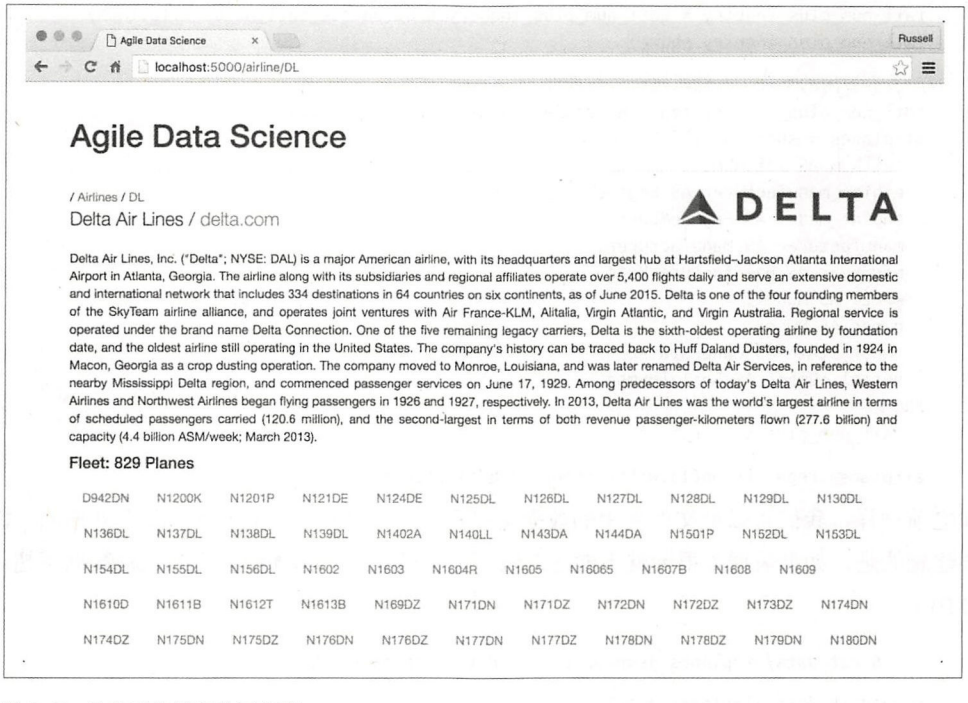


图 6-5　扩充后的航空公司主页

调查飞机（实体）

在第 5 章中，我们发现了一些有意思的数据，现在我们再回头看看。让我们从在中间数据集中存储扩充的飞机数据入手，如 *ch06/prepare_airplanes.py* 所示：

```
# 读取 FAA 航空器注册编号查询记录
faa_tail_number_inquiry = spark.read.json('data/faa_tail_number_inquiry.jsonl')
faa_tail_number_inquiry.show()

# 求记录总数
faa_tail_number_inquiry.count()

# 读取去重的机尾编号表
unique_tail_numbers = spark.read.json('data/tail_numbers.jsonl')
unique_tail_numbers.show()

# 把机尾编号表与查询进行连接
tail_num_plus_inquiry = unique_tail_numbers.join(
  faa_tail_number_inquiry,
  unique_tail_numbers.TailNum == faa_tail_number_inquiry.TailNum,
)
tail_num_plus_inquiry = tail_num_plus_inquiry.drop(unique_tail_numbers.TailNum)
tail_num_plus_inquiry.show()

# 丢弃没用的字段，存储机尾编号与查询
tail_num_plus_inquiry.registerTempTable("tail_num_plus_inquiry")
airplanes = spark.sql("""SELECT
  TailNum AS TailNum,
  engine_manufacturer AS EngineManufacturer,
  engine_model AS EngineModel,
  manufacturer AS Manufacturer,
  mfr_year AS ManufacturerYear,
  model AS Model,
  owner AS Owner,
  owner_state AS OwnerState,
  serial_number AS SerialNumber
FROM
  tail_num_plus_inquiry""")

airplanes.repartition(1).write.json('data/airplanes.json')
```

和之前一样，我们可以把文件夹中的数据复制到单个文件中，以便在 Spark 之外访问。需要注意的是，如果数据量很大就不能这么做了，不过在这个例子中我们的飞机数据不超过 1MB：

```
$ cat data/airplanes.json/part-* >> data/airplanes.jsonl

$ head -5 data/airplanes.jsonl

{
  "TailNum": "N933EV",
  "EngineManufacturer": "GE",
```

```
    "EngineModel": "CF34 SERIES",
    "Manufacturer": "BOMBARDIER INC",
    "ManufacturerYear": "2005",
    "Model": "CL-600-2B19",
    "Owner": "DELTA AIR LINES INC",
    "OwnerState": "GEORGIA",
    "SerialNumber": "8022"
  }
```

我们先以提出问题开始我们的分析：波音公司（Boeing，简称波音）和空中客车公司（Airbus，简称空客），这两家制造商在美国的商业机队中到底哪一家制造的飞机更多？代码如 *ch06/analyze_airplanes.py* 所示：

```
airplanes = spark.read.json('data/airplanes.json')

# 各家制造商制造了多少架飞机？
airplanes.registerTempTable("airplanes")
manufacturer_counts = spark.sql("""SELECT
  Manufacturer,
  COUNT(*) AS Total
FROM
  airplanes
GROUP BY
  Manufacturer
ORDER BY
  Total DESC, Manufacturer"""
)
manufacturer_counts.show(30) # 显示前 30 条记录
```

注意，我们排序时同时使用了总数（`Total`）和制造商（`Manufacturer`）。记住：始终要用一些像这样的附带的“拆分”排序键，使结果在多次重复执行中可以保证一致。如果不指定额外的排序字段，查询结果的顺序就没有被完全定死，具体的顺序取决于 SQL 解释器。执行结果如下：

```
+--------------------+-----+
|        Manufacturer|Total|
+--------------------+-----+
|              BOEING| 2095|
|              AIRBUS|  550|
|      BOMBARDIER INC|  460|
|    AIRBUS INDUSTRIE|  451|
|             EMBRAER|  366|
|   MCDONNELL DOUGLAS|  122|
|MCDONNELL DOUGLAS...|  105|
|         EMBRAER S A|   47|
...
```

有趣的是，波音的飞机数量远超空客，比值大概是 4:1！我原先并不知道，以为它们的数量差距应该要小得多。然而，我实际想了解的是各家飞机制造商的市场占有率（而不是人工计算比例）。也就是说，我希望看到的是以百分比呈现的数据。

SQL 嵌套查询 vs. 数据流编程

这个例子可以很好地解释 SQL 嵌套查询与数据流编程之间的区别。SQL 是声明式的，使用 SQL 你只需要定义你要什么，而不用管 SQL 会如何做。而命令式的数据流编程则需要一步一步计算连接的数据并组合成数据流。

首先，我们使用命令式的数据流编程实现百分比计算，然后再使用声明式的 SQL 嵌套查询实现同样的功能。你会发现在本例中，嵌套查询的方式更为方便，但是嵌套查询的作用也是有限的——嵌套查询很快就会显得很难看懂。最好的方式是创建一系列简单的 SQL 查询或数据流编程语句，然后把它们组合起来进行所需的计算，而不是在一个巨大的多层嵌套查询中完成计算。

不使用嵌套查询的数据流编程

Spark SQL 直到 2.0 版本才支持了嵌套查询。有了各个制造商的飞机数量，我们需要求出总的飞机数量，与已有的各制造商总数连接，然后用各制造商总数除以总飞机数量。我们需要重复使用前面例子中求得的表 `manufacturer_counts`：

```
# 总共有多少架飞机?
total_airplanes = spark.sql(
  """SELECT
  COUNT(*) AS OverallTotal
  FROM airplanes"""
)
print("Total airplanes: {}".format(total_airplanes.collect()[0].OverallTotal))

mfr_with_totals = manufacturer_counts.join(total_airplanes)
mfr_with_totals = mfr_with_totals.rdd.map(
  lambda x: {
    'Manufacturer': x.Manufacturer,
    'Total': x.Total,
    'Percentage': round(
      (
        float(x.Total)/float(x.OverallTotal)
      )*100,
      2
    )
  }
)
mfr_with_totals.toDF().show()
```

所得结果如下：

```
+--------------------+----------+-----+
|        Manufacturer|Percentage|Total|
+--------------------+----------+-----+
|              BOEING|     49.04| 2095|
|              AIRBUS|     12.87|  550|
|      BOMBARDIER INC|     10.77|  460|
|    AIRBUS INDUSTRIE|     10.56|  451|
|             EMBRAER|      8.57|  366|
|   MCDONNELL DOUGLAS|      2.86|  122|
|MCDONNELL DOUGLAS...|      2.46|  105|
|         EMBRAER S A|       1.1|   47|
...
```

这显然是一种冷门的计算百分比的方法，不过这也为如何使用数据流编程解决复杂问题提供了示例。

Spark SQL 中的子查询

子查询很好用，使用子查询计算飞机制造商的份额百分比很简单：

```
relative_manufacturer_counts = spark.sql("""SELECT
  Manufacturer,
  COUNT(*) AS Total,
  ROUND(
    100 * (
      COUNT(*)/(SELECT COUNT(*) FROM airplanes
    ),
    2
  ) AS PercentageTotal
  FROM
    airplanes
GROUP BY
  Manufacturer
ORDER BY
  Total DESC, Manufacturer"""
)
relative_manufacturer_counts.show(30) # 显示前 30 条记录
```

所得结果与前一节的结果完全一样：

```
+--------------------+-----+---------------+
|        Manufacturer|Total|PercentageTotal|
+--------------------+-----+---------------+
|              BOEING| 2095|          49.04|
|              AIRBUS|  550|          12.87|
|      BOMBARDIER INC|  460|          10.77|
|    AIRBUS INDUSTRIE|  451|          10.56|
|             EMBRAER|  366|           8.57|
|   MCDONNELL DOUGLAS|  122|           2.86|
|MCDONNELL DOUGLAS...|  105|           2.46|
|         EMBRAER S A|   47|            1.1|
...
```

创建飞机主页

现在我想以图表的形式在网页上查看这些数据，这意味着我们要找个地方放置图表。是时候创建 */airplanes* 作为飞机主页了，在这个页面上可以对机队进行整体分析。

我们先给 */airplanes* 创建 Flask 控制器。如 *ch06/web/report_flask.py* 所示，我们直接从 Mongo 中读取数据，并将数据传给模板 *all_airplanes.html*：

```
@app.route("/airplanes")
@app.route("/airplanes/")
def airplanes():
  mfr_chart = client.agile_data_science.manufacturer_totals.find_one()
  return render_template('all_airplanes.html',mfr_chart=mfr_chart)
```

初始版的 *all_airplanes.html* 及其效果页面都很简单：

```
{% extends "layout.html" %}
{% block body %}
  <!-- 导航栏 -->
  / <a href="/airplanes">Airplanes</a>

  <p class="lead">
    <!-- 页面标题 -->
    US Commercial Fleet
  </p>
{% endblock %}
```

结果飞机主页见图 6-6。

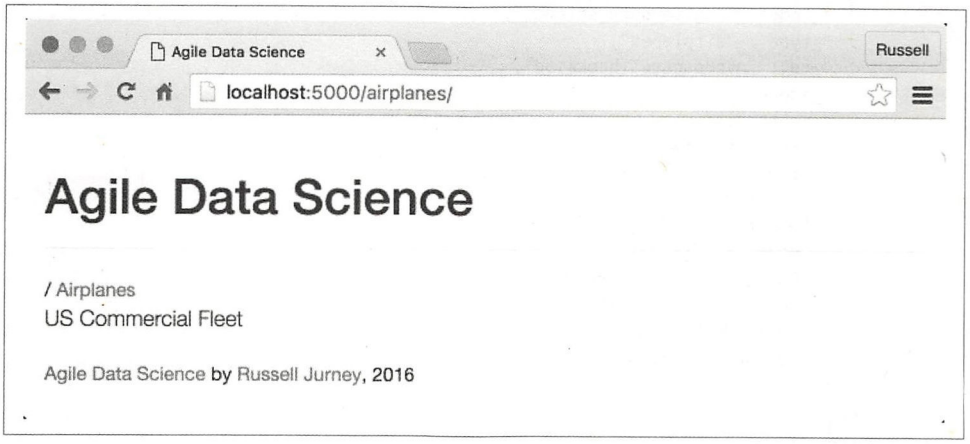


图 6-6　飞机主页

在飞机页面上添加搜索

/airplanes 页面是一个提供飞机数据搜索的好地方。要实现搜索功能，首先需要通过 PySpark 把飞机数据写入 Elasticsearch：

```
# 读取飞机记录
airplanes = spark.read.json("data/airplanes.json")
airplanes.show()

airplanes.write.format("org.elasticsearch.spark.sql")\
  .option("es.resource","agile_data_science/airplanes")\
  .mode("overwrite")\
  .save()
```

然后，通过一条简单的查询就可验证文件是否已写出：

```
curl -XGET 'localhost:9200/agile_data_science/airplanes/_search?q=*'
```

可以得到 4,272 条结果：

```
{
  "took": 3,
  "timed_out": false,
  "_shards": {
  "total": 5,
  "successful": 5,
  "failed": 0
 },
 "hits": {
   "total": 4272,
   "max_score": 1,
   "hits": [
     {
```

```
            "_index": "agile_data_science",
            "_type": "airplanes",
            "_id": "AVbYpad6tuTlhookKT6d",
            "_score": 1,
            "_source": {
              "EngineManufacturer": "ROLLS-ROYC",
              "EngineModel": "RB.211 SERIES",
              "Manufacturer": "BOEING",
              "ManufacturerYear": "1999",
              "Model": "757-224",
              "Owner": "UNITED AIRLINES INC",
              "OwnerState": "ILLINOIS",
              "SerialNumber": "29284",
              "TailNum": "N41135"
            }
          },
          ...
        ]
      }
    }
```


现在，我们可以为 /airplanes 控制器增加搜索功能了。回忆一下第 5 章中我们为航班增加搜索功能时的做法，用了好几页的代码来实现。这一次我们要构建一个可重用的组件，用于在 Flask 应用程序中搜索和显示记录。

代码 vs. 配置参数

对于在分解代码之前有多少代码重复是可接受的，有很多不同的观点，根据具体的环境和编程规则也会有很大差异。与通用软件工程相比，数据科学对丑陋、重复的代码具有更高的容忍度。这是因为数据科学家写的大多数代码在运行后会立即被丢弃。大多数数据科学家都做得很好，能把所有的代码都提交到一个代码仓库中（绝对必须！）。但是，如果代码没有被丢弃，而且我们还要分享结果，就像我们的应用程序中的代码一样，代码整洁就凸显其重要性了。

这是我们第二次需要实现搜索，而在我们发现需要大量重复代码时，应该停下来想一想是不是避免一些重复代码也可以实现类似的功能。（因为数据科学中很多代码是会被扔掉的，所以我们应该等到确实需要重复代码的时候再来进行重构，使代码更为通用，就像此处所做的一样。）

确定了要把代码改得更通用之后，我们需要把代码分为两个组成部分：算法和配置参数。算法定义了我们要做什么；配置参数则定义了其中具体的一个算法。

拿极端例子来说，我们可以在编程中去掉所有的重复代码，这样一切都变成了配置文件和根据配置文件实现的算法。然而这个极端情况是不可维护的，也是和数据科学家的工作负

担不符的，因为这样我们需要同时记住很多东西，无法深入各个组件进行修改和使用。和敏捷数据科学一样，我们选择折中的方法，只移除最明显的重复代码，而不把所有逻辑都弄成最一般化、可以复用的形式。

配置搜索框

搜索的配置是很简单的。配置项包括我们要搜索和展示的字段，以及要在用户界面中显示时使用的标签：

```
search_config = [
  {'field': 'TailNum', 'label': 'Tail Number'},
  {'field': 'Owner'},
  {'field': 'OwnerState', 'label': 'Owner State'},
  {'field': 'Manufacturer'},
  {'field': 'Model'},
  {'field': 'ManufacturerYear', 'label': 'MFR Year'},
  {'field': 'SerialNumber', 'label': 'Serial Number'},
  {'field': 'EngineManufacturer', 'label': 'Engine MFR'},
  {'field': 'EngineModel', 'label': 'Engine Model'}
]
```

编程实现构建 Elasticsearch 查询

我们的分页模块依然能用，现在给它加上新的配置项 AIRPLANE_RECORDS_PER_PAGE：

```
# 分页参数
start = request.args.get('start') or 0
start = int(start)
end = request.args.get('end') or config.AIRPLANE_RECORDS_PER_PAGE
end = int(end)

# 配置导航路径和起点偏移
nav_path = search_helpers.strip_place(request.url)
nav_offsets = search_helpers.get_navigation_offsets(
start, end, config.AIRPLANE_RECORDS_PER_PAGE
)
```

有了搜索配置，我们只需要定义 Elasticsearch 查询的基础，然后根据收到的搜索参数把查询语句补充完整。我们的基础查询如下所示：

```
# 构建 Elasticsearch 查询的基础
query = {
  'query': {
    'bool': {
      'must': []}
  },
  'sort': [
```

仅供非商业用途或交流学习使用

```
      {'Owner': {'order': 'asc', 'ignore_unmapped': True}},
      {'Manufacturer': {'order': 'asc', 'ignore_unmapped' : True} },
      {'ManufacturerYear': {'order': 'asc', 'ignore_unmapped' : True} },
      {'SerialNumber': {'order': 'asc', 'ignore_unmapped' : True} },
      '_score'
    ],
    'from': start,
    'size': config.AIRPLANE_RECORDS_PER_PAGE
  }
```

用如下方式把它参数化：

```
arg_dict = {}
for item in search_config:
  field = item['field']
  value = request.args.get(field) arg_dict[field] = value
  if value:
    query['query']['bool']['must'].append({'match': {field: value}})
```

还用老方法提交查询：

```
# 查询 Elasticsearch，处理获取结果和记录总数
results = elastic.search(query)
airplanes, airplane_count = search_helpers.process_search(results)
```

在调用模板渲染时，我们要额外指定参数 search_config 和 arg_dict，这样可以在模板中生成相应的内容：

```
# 把搜索配置参数保存到表单模板中
return render_template(
  'all_airplanes.html',
  search_config=search_config,
  args=arg_dict,
  airplanes=airplanes,
  airplane_count=airplane_count,
  nav_path=nav_path,
  nav_offsets=nav_offsets
)
```

我们的模板 *all_airplanes.html* 是继承自 *search.html* 的。根据 search_config 和请求参数，可以编程构建出在 *search.html* 中声明的所有内容。我们可以重用这部分代码来构建任何搜索的控制器：

```
{% extends "layout.html" %}
{% block body %}

  / <a href="/airplanes">Airplanes</a>

  <p class="lead">
    <!-- 页面标题 -->
    US Commercial Fleet
  </p>
```

仅供非商业用途或交流学习使用

```
  <!-- 根据 search_config 和请求参数生成表单 -->
  <form action="/airplanes"method="get">
    {% for item in search_config %}
      {% if 'label' in item %}
        <label for="{{item['field']}}">{{item['label']}}</label>
      {% else %}
        <label for="{{item['field']}}">{{item['field']}}</label>
      {% endif %}
        <input name="{{item['field']}}"
                value="{{args[item['field']] if args[item['field']] else ''}}">
        </input>
    {% endfor %}
    <button type="submit"class="btn btn-xs btn-default"style="height: 25px">
      Submit
    </button>
  </form>

  <table class="table table-condensed table-striped">

    <!-- 根据 search_config 创建表头 -->
    <thead>
      {% for item in search_config %}
      {% if 'label' in item %}
        <th>{{item['label']}}</th>
      {% else %}
        <th>{{item['field']}}</th>
      {% endif %}
      {% endfor %}
    </thead>

<!--
  填充表内容，根据飞机记录填充各个 <tr>，
  根据 search_config 填充各个 <td>
-->
    <tbody>
      {% for airplane in airplanes %}
      <tr>
        {% for item in search_config %}
          <td>{{airplane[item['field']]}}</td>
        {% endfor %}
      </tr>
      {% endfor %}
    </tbody>
  </table>

{% import "macros.jnj" as common %}
{% if nav_offsets and nav_path -%}
   {{ common.display_nav(nav_offsets, nav_path, airplane_count)|safe }}
{% endif -%}
{% endblock %}
```

仅供非商业用途或交流学习使用

创建飞机制造商的条形图

现在有地方可以放置图表了，让我们开始做图表吧！

继续执行脚本 *ch06/analyze_airplanes.py*，我们把图表要用的数据存储到 Mongo 中：

```
#
# 获取网页上要用的数据
#
relative_manufacturer_counts = relative_manufacturer_counts.rdd.map(
  lambda row: row.asDict()
)
grouped_manufacturer_counts = relative_manufacturer_counts.groupBy(lambda x: 1)

# 存到 Mongo 的表 airplanes_per_carrier 中
import pymongo_spark pymongo_spark.activate() grouped_manufacturer_counts.saveToMongoDB(
  'mongodb://localhost:27017/agile_data_science.airplane_manufacturer_totals'
)
```

接下来，确认数据在 Mongo 中：

```
>db.manufacturer_totals.find()
{
  "_id":1,
  "maxindex":35,
  "data":[
     {
       "PercentageTotal":49.04,
       "Manufacturer":"BOEING",
       "Total":2095
     },
     {
       "PercentageTotal":12.87,
       "Manufacturer":"AIRBUS",
       "Total":550
     },
     ...
  ]
}
```

再接下来就和第 5 章中制作条形图类似了。我们在 *report_flask.py* 中增加一个控制器，从 Mongo 中读取表的数据，以 JSON 格式返回：

```
@app.route("/airplanes/chart/manufacturers.json")
@app.route("/airplanes/chart/manufacturers.json")
def airplane_manufacturers_chart():
  mfr_chart = client.agile_data_science.manufacturer_totals.find_one()
  return json.dumps(mfr_chart)
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

然后，我们修改 *all_airplanes.html* 模板来调用 *airplane.js*，用它来画图表。

这次我们需要自定义条形图的 *x* 轴和 *y* 轴，因此我们先找个同时包括 *x* 轴和 *y* 轴的例子。Mike Bostock 的示例（条形图 IIIc）简单又明白。我们先给页面命名，然后调用画图脚本 *airplane.js*：

```
<div>
  <p class="lead">Total Flights by Month</p>
  <div id="chart"><svg class="chart"></svg></div>
</div>
<script src="/static/airplane.js"></script>
```

/static/airplane.js 也做了一些变化，使得表中的数据能够转为图表，如下面的代码所示。除了飞机总数（`Total`）及制造商（`Manufacturer`）字段，还要将 `data.data` 字段传递过去，其他 地方就没有再修改了，只有图表的维度变了：

```
var margin = {top: 20, right: 30, bottom: 30, left: 40},
    width = 900 - margin.left - margin.right,
    height = 300 - margin.top - margin.bottom;

var x = d3.scale.ordinal()
    .rangeRoundBands([0, width], .1);
var y = d3.scale.linear()
    .range([height, 0]);

var xAxis = d3.svg.axis()
    .scale(x)
    .orient("bottom");

var yAxis = d3.svg.axis()
    .scale(y)
    .orient("left");

var chart = d3.select(".chart")
    .attr("width", width + margin.left + margin.right)
    .attr("height", height + margin.top + margin.bottom)
    .append("g")
    .attr("transform", "translate("+ margin.left + ","+ margin.top + ")");

console.log("HELLO");
d3.json("/airplanes/chart/manufacturers.json", function(error, data) {
    var data = data.data;

    x.domain(data.map(function(d) { return d.Manufacturer; }));
    y.domain([0, d3.max(data, function(d) { return d.Total; })]);

    chart.append("g")
        .attr("class", "x axis")
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

```
            .attr("transform", "translate(0,"+ height + ")")
            .call(xAxis);
        chart.append("g")
            .attr("class", "y axis")
            .call(yAxis);
    chart.selectAll(".bar")
        .data(data)
        .enter().append("rect")
        .attr("class", "bar")
        .attr("x", function(d) { return x(d.Manufacturer); })
        .attr("y", function(d) { return y(d.Total); })
        .attr("height", function(d) { return height - y(d.Total); })
        .attr("width", x.rangeBand());
    });
```

经过一番操作，我们获得了一个好看的图表（见图 6-7）。

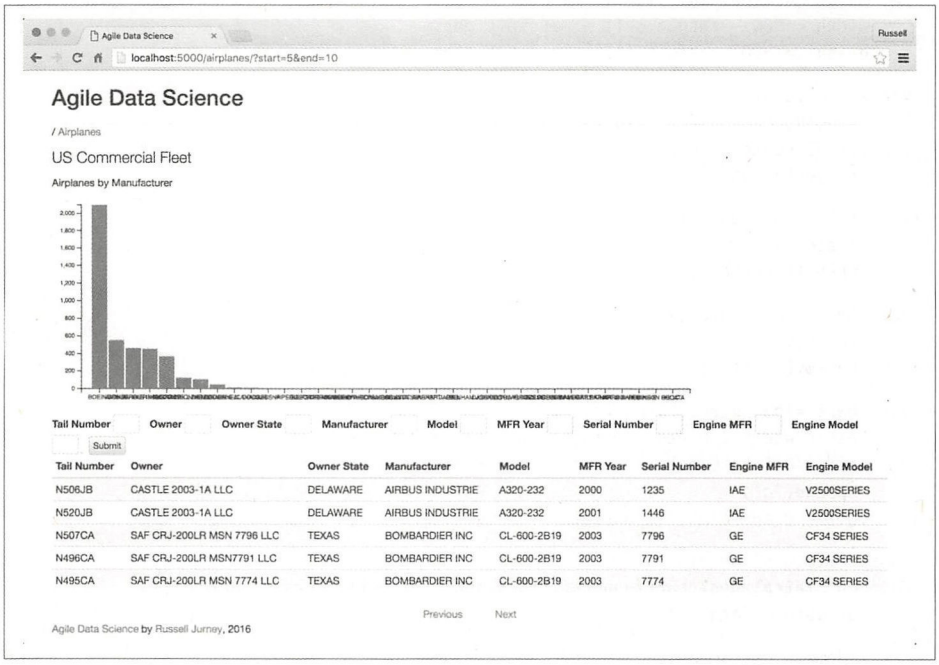


图 6-7　加上了新图表的航空公司主页

对飞机制造商条形图进行迭代

等一下，不对劲！还记得吗？我们说过迭代是必不可少的。我们需要对图表进行一番调试，并从图表中推断出我们还要做什么。为什么条形图的条这么细？为什么都紧紧靠在最左边？

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

回忆一下，我们是根据飞机总数的降序来存储数据的：

```
relative_manufacturer_counts = spark.sql("""SELECT
  Manufacturer,
  COUNT(*) AS Total,
  ROUND(
    100 * (
      COUNT(*)/(SELECT COUNT(*) FROM airplanes)
    ),
    2
  ) AS PercentageTotal
FROM
  airplanes
GROUP BY
  Manufacturer
ORDER BY
  Total DESC, Manufacturer"""
)
```

这意味着最大的值在左边，而最小的值在右边……所以结果肯定是有太多的小值，导致图表变成这样而不便阅读！我们可以通过删除一些较小的值来改进图表，毕竟它们无关紧要。请注意，情况并不总是如此的，所以在删除任何数据之前都要想清楚！

我们可以通过使用 SQL 的 `LIMIT` 从句重算数据，修正我们的图表。首先，我们需要从 Mongo 中删除老的数据：

```
mongo agile_data_science
> db.airplane_manufacturer_totals.drop()
```

现在，回到 *analyze_airplanes.py* 中，加上 `LIMIT 10` 来获取前 10 大制造商：

```
relative_manufacturer_counts = spark.sql("""SELECT
  Manufacturer,
  COUNT(*) AS Total,
  ROUND(
    100 * (
      COUNT(*)/(SELECT COUNT(*) FROM airplanes)
    ),
    2
  ) AS PercentageTotal
FROM
  airplanes
GROUP BY
  Manufacturer
ORDER BY
  Total DESC, Manufacturer
LIMIT 10"""
)
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

运行新脚本，把数据存入 Mongo，我们得到的图表清晰显示出波音在市场中的统治地位，其他一些制造商紧随其后（见图 6-8）。注意，我们还创建了 `truncate` 函数来缩短 x 轴上制造商的名字，使得这些标签不至于重叠。我们在 `xAxis` 对象的 `tickFormat` 方法中调用这个函数：

```
function truncate(d, l) {
    if(d.length >l)
        return d.substring(0,l)+'...';
    else
        return d;
}

var xAxis = d3.svg.axis()
    .scale(x)
    .orient("bottom")
    .tickFormat(function(d) {
        return truncate(d, 14);
        });
```

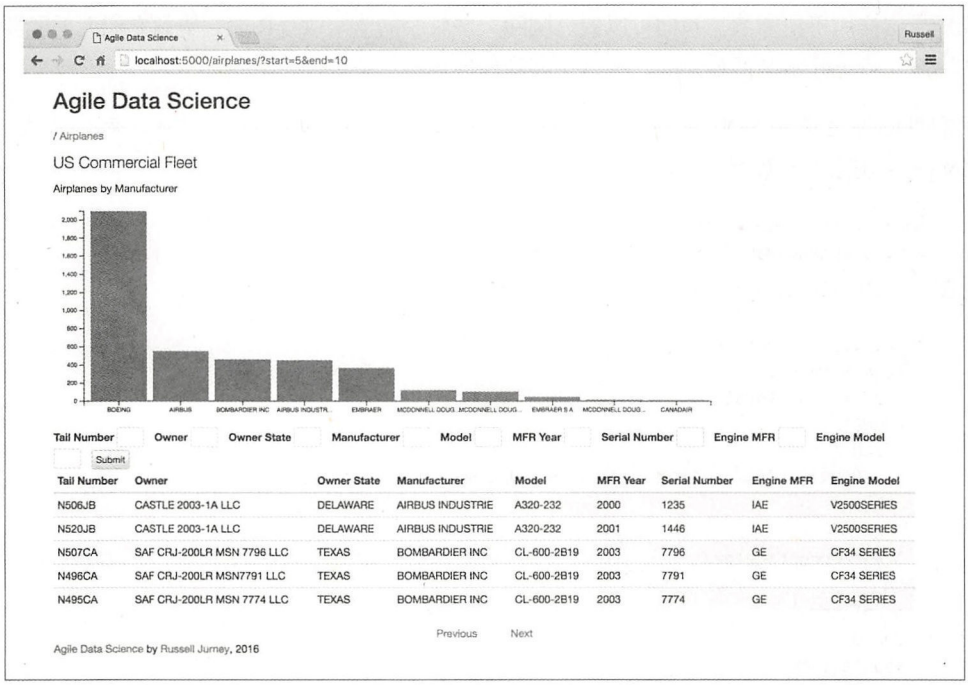


图 6-8　包含改进过图表的航空公司主页

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

实体解析：新一轮图表迭代

然而，此时的图表还有一个问题——列名有重复，导致空客、麦克唐纳·道格拉斯公司（McDonnel Douglas，简称麦道）、巴西航空工业公司（Embraer）等制造商的数值不准确。我们又要迭代了！这次我们要使用实体解析。

30 秒实体解析

我们遇到的问题是同一制造商的名字在飞机注册中使用了多种形式。处理这一问题就叫作实体解析，Lise Getoor 和 Ashwin Machanavajjhala 写的实体解析指南中把它定义为“识别并合并真实世界中同一对象的多种表现形式”。实体解析就是把“空中客车（AIRBUS）”和“空中客车公司（AIRBUS INDUSTRIE）”识别为同一个主体的过程。

实体解析有很多种方法，包括使用统计推断的复杂方式。我们仅仅探索一种简单的启发式的方式，因为在本例中这已经足够好用。不要因为好奇就分心去尽量使用机器学习和统计学技术。还是把好奇心留给结果吧。

在 PySpark 中解析制造商

首先查看飞机表中制造商字段的不同值。我们可以使用 SQL 语句 SELECT DISTINCT（Manufacturer）AS Manufacturer 并以 ORDER BY Manufacturer 子句进行排序，然后就会看到近似的记录都靠在一起。接下来，我们就只需要以左对齐的方式打印数据，看看我们得到了什么！

```
airplanes = spark.read.json('data/airplanes.json')

airplanes.registerTempTable("airplanes")
manufacturer_variety = spark.sql(
"""SELECT
  DISTINCT(Manufacturer) AS Manufacturer
FROM
  airplanes
ORDER BY
  Manufacturer"""
)
manufacturer_variety_local = manufacturer_variety.collect()

# 我们要以左对齐的方式打印
for mfr in manufacturer_variety_local:
print(mfr.Manufacturer)
```

结果列表能够让我们轻松查看制造商的不同取值：

仅供非商业用途或交流学习使用

```
...
AIRBUS
AIRBUS INDUSTRIE
...
EMBRAER
EMBRAER S A
...
GULFSTREAM AEROSPACE
GULFSTREAM AEROSPACE CORP
...
MCDONNELL DOUGLAS
MCDONNELL DOUGLAS AIRCRAFT CO
MCDONNELL DOUGLAS CORPORATION
...
```

事实证明，并没有多少不同的取值：只有 35 个而已。可以手动完成对这些记录中制造商字段的调整，用简单的表格详细说明两列中的匹配关系。一列包含原始值，另一列则包含要映射到的值（你选择的“标准值”）。你可以对这个表进行左外连接，如果存在匹配的话，替换该字段的值，这样就可以让记录使用通用的标识符。

如果你在工作中遇到了一个只有 35 个值的字段，那么给自己省点事吧：手动把表写为 CSV 格式，然后用 Spark 读取并进行表连接。在这里，我们将进一步说明如何以自动化的方式创建这样一个映射表，以及如何连接并应用映射表。我们这样做是为了给你如何解决问题并尽量摆脱这种情况的经验，而无须采用更复杂的统计学技术（那样会很浪费时间的）。

更复杂的方法是检查数据，看看我们是不是可以推导出一个规则来决定记录是否相同。在查看重复内容时，似乎每当有重复的时候，字符串的起始处都有很多重叠。这在未处理的公司名称中很常见，“Incorporated”作为后缀可以缩写为“Inc,”“Inc.,”“INC,”“Corp,”等。于是我们可以制定一个策略：如果记录中该字段的值在字符串的起始处有超过 N 个一样的字符，就认为它们是相同的。我们将选择最长的公共子字符串作为这些记录的“标准”值，并使用这一规则创建我们的映射表。

要使用这种策略，我们需要比较制造商所有不同的值。这对于 35 个值是可行的，但请记住实体解析并不是每次都可以这么做的。有时不可能两两比较所有记录，因为去重记录总数的平方太大，即使用 Spark 也不行！在这种情况下我们只解析一个字段，只使用单个字段的去重值域，这样维度不会过高。当记录用许多字段来区分时，去重记录的数量便会爆炸性增长。这种情况（幸好）不在本书的讨论范围之内，但是我使用 Swoosh 算法（*http://*

仅供非商业用途或交流学习使用

infolab.stanford.edu/serf/swoosh_vldbj.pdf）应对这种情况有过不错的经验，这种算法是在斯沃福的 SERF（*http://infolab.stanford.edu/serf/*）项目中实现的。

请参考 *ch06/resolve_airplane_manufacturers.py*。在这里，我们假设字符串开头五个字符相同的为同一个值，据此为制造商字段的相似取值创建映射表。请注意，这个假设很简单，并不适用于大多数的数据集。不过，它展示了如何通过学习数据集和实际查看排序去重的数据来摆脱泥潭。

在上一段示例代码中，我们计算得到 `tmanufacturer_variety`，接着来看一下此处描述计算步骤的注释：

```
# 找到两个字符串开头最长的公共子字符串
def longest_common_beginning(s1, s2):
  if s1==s2:
    return s1
  min_length = min(len(s1), len(s2))
  i=0
  while i <min_length:
    if s1[i] == s2[i]:
      i+=1
    else:
      break
  return s1[0:i]

# 比较两个制造商，返回描述结果的元组
def compare_manufacturers(mfrs):
  mfr1 = mfrs[0]
  mfr2 = mfrs[1]
  lcb = longest_common_beginning(mfr1, mfr2)
  len_lcb = len(lcb)
  record = {
    'mfr1': mfr1,
    'mfr2': mfr2,
    'lcb': lcb,
    'len_lcb': len_lcb,
    'eq': mfr1 == mfr2
  }
  return record

# 把所有不一样的制造商字段两两配对进行比较
comparison_pairs = manufacturer_variety.join(manufacturer_variety)

# 进行比较
comparisons = comparison_pairs.rdd.map(compare_manufacturers)

# 开头有超过 5 个相同字符的作为匹配项
matches = comparisons.filter(lambda f: f['eq'] == False and f['len_lcb'] >5)
```


仅供非商业用途或交流学习使用

```
#
# 我们现在为重复键创造了从原始值到我们要用的值的映射表
#

# 1) 按最长相同起始字符（'lcb'）对匹配项进行分组
common_lcbs = matches.groupBy(lambda x: x['lcb'])

# 2) 把两边的原始值与 'lcb' 一起拿出来
mfr1_map = common_lcbs.map(
  lambda x: [(y['mfr1'], x[0]) for y in x[1]]).flatMap(lambda x: x)

mfr2_map = common_lcbs.map(
  lambda x: [(y['mfr2'], x[0]) for y in x[1]]).flatMap(lambda x: x)

# 3) 合并比较记录的两边
map_with_dupes = mfr1_map.union(mfr2_map)

# 4) 去除重复项
mfr_dedupe_mapping = map_with_dupes.distinct()

# 5) 把映射表转为 dataframe 来和飞机 dataframe 进行连接
mapping_dataframe = mfr_dedupe_mapping.toDF()

# 6) 给映射表字段设置列名
mapping_dataframe.registerTempTable("mapping_dataframe")
mapping_dataframe = spark.sql(
  "SELECT _1 AS Raw, _2 AS NewManufacturer FROM mapping_dataframe"
)
```

现在可以使用我们创建的映射表了。注意，因为记录不多，所以这张表可以手动做出来，在这种情况下，你应该把映射表读取为 CSV（然后运行如下代码块）：

```
# 用左外连接
airplanes_w_mapping = airplanes.join(
  mapping_dataframe,
  on=airplanes.Manufacturer == mapping_dataframe.Raw,
  how='left_outer'
)
# 现在根据需要把 Manufacturer 替换为 NewManufacturer
airplanes_w_mapping.registerTempTable("airplanes_w_mapping")
resolved_airplanes = spark.sql("""SELECT
  TailNum,
  SerialNumber,
  Owner,
  OwnerState,
  IF(NewManufacturer IS NOT null,NewManufacturer,Manufacturer) AS Manufacturer,
  Model,
  ManufacturerYear,
  EngineManufacturer,
```


仅供非商业用途或交流学习使用

```
  EngineModel
FROM
  airplanes_w_mapping""")

# 存下来以便以后使用，替换 airplanes.json
resolved_airplanes.repartition(1).write.mode("overwrite") \
  .json("data/resolved_airplanes.json")
```

同样，为了方便，我们创建出单个 JSON 行文件：

```
cat data/resolved_airplanes.json/part* >> data/resolved_airplanes.jsonl
```

现在让我们更新图表吧！

更新图表

我们需要运行 *ch06/analyze_airplanes_again.py*，这是由 *ch06/analyze_airplanes.py* 复制过来的，替换了我们解析好的飞机表。做好之后，访问 */airplanes* 来查看更新后的图表（见图 6-9）。

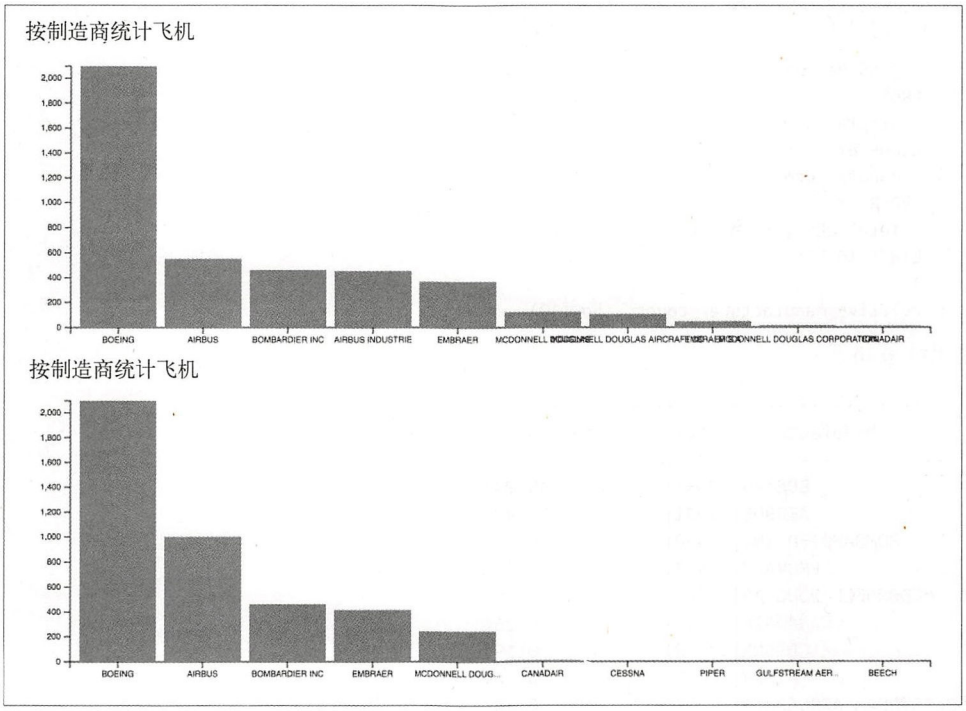


图 6-9　原先的图表（上方）与去重后的图表（下方）

在我们去除了重复的制造商之后，图表的样子也发生了一些变化！空客不像我们先前看到的那样远远落后于波音了。现在我很想知道每个制造商根据新的 `airplains` 表求出的精确的市场份额。

重看波音和空客的对比

为了求出波音和空客的精确市场份额，我们再运行一遍百分比计算程序，如 *ch06/analyze_airplanes_again.py* 所示：

```
airplanes = spark.read.json('data/resolved_airplanes.json')
airplanes.registerTempTable("airplanes")

relative_manufacturer_counts = spark.sql("""SELECT
  Manufacturer,
  COUNT(*) AS Total,
  ROUND(
    100 * (
      COUNT(*)/(SELECT COUNT(*) FROM airplanes)
    ),
    2
  ) AS PercentageTotal
FROM
  airplanes
GROUP BY
  Manufacturer
ORDER BY
  Total DESC, Manufacturer
LIMIT 10"""
)
relative_manufacturer_counts.show(10)
```

输出结果如下：

```
+--------------------+-----+---------------+
|        Manufacturer|Total|PercentageTotal|
+--------------------+-----+---------------+
|              BOEING| 2095|          49.04|
|              AIRBUS| 1001|          23.43|
|      BOMBARDIER INC|  460|          10.77|
|             EMBRAER|  413|           9.67|
|   MCDONNELL DOUGLAS|  241|           5.64|
|            CANADAIR|   11|           0.26|
|              CESSNA|    8|           0.19|
|               PIPER|    7|           0.16|
|GULFSTREAM AEROSPACE|    6|           0.14|
|               BEECH|    5|           0.12|
+--------------------+-----+---------------+
```

结果波音的市场占有率为 49%，而空客为 23.4%。选择波音！（或者，如果你在欧洲，选择空客！）

整洁：实体解析的作用

原始数据始终是“脏”的。当你深入其中，操作数据、查看数据原始的样子、把它呈现在网页上的表格和图表中，或是提供搜索它的功能时，原始数据的问题就出现了。在操作数据中解决这些问题可以让你清楚地看到真实的趋势。和对于可视化的好处一样，对模型也有好处。这种“整洁”可以让你有效地在数据价值金字塔的下一层构建统计学模型：预测（*prediction*）。

本章小结

下面是对目前为止我们所学内容的总结：

1. 创建有趣的、相互连接的记录。一开始“有趣”的标准较低，随着时间的推移，我们会根据用户反馈、流量分析及评论提高这一标准。

2. 把这些记录作为对象存储到文档存储中，如下所示：

   ```
   key => {property1, property2, links => [key1, key2, key3]}
   ```

   一种方法是分割记录，增加属性使数据更复杂，以避免深度嵌套；另一种方法是直接作为文档存储。只要适用于数据，两种方式都是行之有效的。

3. 优先使用 Flask 或 Sinatra 这样的轻量级网络应用框架，以 JSON 方式传输键 / 值对数据，使用以 JSON 格式返回的文档存储。

在下一章中会使用我们所学到的数据相关的技术，进行一个有实际意义的预测：我们的航班会延误吗？如果会，大概会延误多久？

第 7 章

进行预测

现在我们有了交互式报表，从多个方面展现了我们的数据，下面就要进行我们的第一个预测了。这就是第四个敏捷开发冲刺（见图 7-1）。

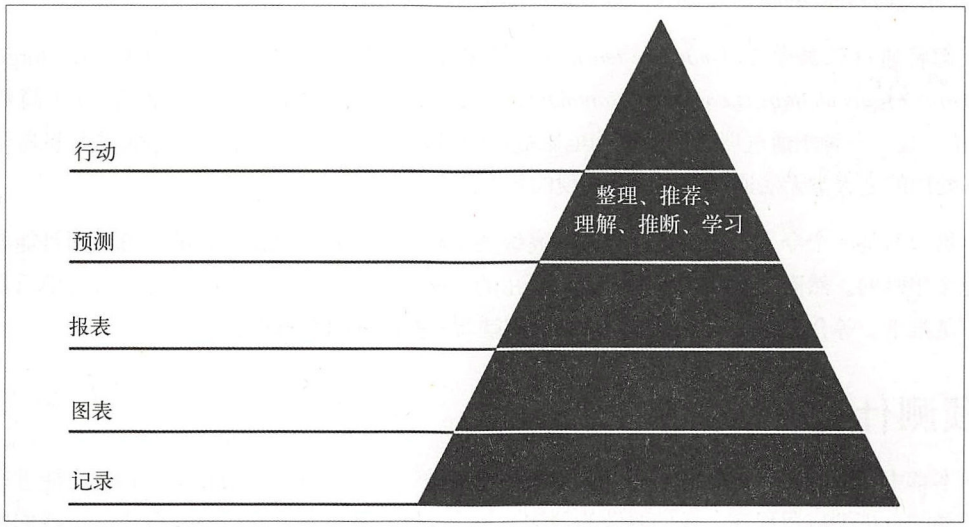


图 7–1　第四层：进行预测

在进行预测的时候，可以使用我们掌握的历史数据对未来进行推测。在这一过程中，我们从对历史数据的批处理转向对未来的实时推测。实际上，本章的任务是使用航班记录的历史数据预测未来的航班情况。

本章的代码示例可以在 *Agile_Data_Code_2/ch07* 中找到。可以通过克隆代码仓库来学习！

```
git clone https://github.com/rjurney/Agile_Data_Code_2.git
```

预测的作用

我们见过很多生活中的预测。有一些预测是根据统计推断的，还有一些则仅仅是专家的意见。在各种预测中，统计推断的运用越来越多了。从天气预报到保险精算师估算保费，从体育博彩中的让分到扑克游戏的胜率，统计预测是当代生活的一部分。有时候预测是准确的，有时候则不是。

举个例子，我在撰写本书时，专家们一致认为唐纳德·特朗普竞选总统是个笑话，即便他不断前进，逐渐领先并最终击败所有的初选对手，在大选前不断逼近希拉里·克林顿。专家们经常得出错误的结论，而 FiveThirtyEight（*http://53eig.ht/2omLMdA*）公司的 Nate Silver 则做出了更准确的大选预测。他使用高级统计模型——538 回归来预测各州大选的结果，然后把这些预测合并到一个在 2008 年和 2012 年都很准确的模型里（尽管最终 Silver 和世界上所有信任美国投票者的理性人士一样，没能预测到特朗普胜选，但公平地说，他预测出的特朗普的胜率为 29%，差不多是别人预测的两倍）。

我们将通过机器学习（*machine learning*）技术使用统计推断进行预测。根据 TechTarget（*http://whatis.techtarget.com/definition/machine-learning*）给出的定义，机器学习（简称 ML）是“一种不通过明确的编程为电脑提供学习能力的人工智能”。另一种说法是机器学习处理的是人类无法通过手工编程实现的复杂任务。

机器学习是一个令人生畏的主题，一门高级的学科方向。掌握机器学习的方方面面可能要花多年时间。然而，多亏了本章中我们要用的一些工具的出现，在实践中上手机器学习并不是难事。等介绍完一些基础知识后，我们就写一些简单的代码感受一下。

预测什么

在本章中，我们要使用机器学习，基于我们已经做了很久可视化的数据集，构建出能进行预测的分析型应用程序。我们要做的预测对每个坐飞机的人都有很大的实际意义。我们要预测航班延误（*fight delay*）。具体地说，我们要预测到达时间的延误，也就是到达目的地机场抵达口时晚了多久。

首先，我们来谈谈预测分析的基础知识。

预测分析导论

维基百科（*https://en.wikipedia.org/wiki/Predictive_analytics*）上说："预测分析涵盖预测建模、机器学习、数据挖掘等一系列统计学技术，通过分析当前和历史的情况，对未来或未知事件进行预测"。

预测分析需要训练数据。训练数据是由我们要预测的样本实体组成的。样本则是由一个以上的特征组成的。因变特征（*dependent feature*）是我们尝试去预测的值。自变特征（*independent feature*）则用来描述我们想预测的事物，并且和因变特征有关（*relate*）。比如，我们用来预测航班延误的训练数据是那些航班延误的原子记录。一次航班就是一条记录，而其延误时间就是这条记录的因变特征。自变特征则是和某次航班有关联的方方面面的内容，换句话说，就是我们在前几章中涉及的所有实体及属性！自变特征是航班的其他属性——比如起飞延误、执飞航空公司、出发城市和终点城市、飞行日期（星期几或一年中的哪一天）等。

我们已经通过分析数据，增进了对航班特征的理解。我们都很熟悉航班延误、航班及其要素：飞机、航空公司、机场等。这可以让我们更高效地进行特征工程（*feature engineering*）——进行预测的关键环节。以交互式可视化和探索性数据分析作为特征工程的一部分，是敏捷数据科学的核心。这两件事驱动并组织着我们的工作。

基础都已经打好了，接下来我们来学习预测的机制吧。

进行预测

大多数预测都可以通过两种方式来实现：回归和分类。回归（*regression*）把由特征组成的样本作为输入，给出数值输出。分类（*classification*）把样本作为输入，把分类作为输出。统计预测的输入是样本集，它让机器能够学习，我们把它称为训练数据（*training data*）。

究竟是使用回归还是分类，取决于我们的业务需求。返回值的类型是我们使用哪个的决定性因素。如果要预测的结果是一个连续变量，则使用回归。如果要预测的是一个名义变量或分类变量，则使用分类。

然而，考虑到我们向用户提供预测的接口，具体的选择可能要复杂得多。比如，如果我们要卖预测航班延误的 API，那么我们可能要用回归来计算一个数值结果，而如果要把航班延误情况直接在手机应用中向用户展示，那么考虑到易用性，用分类来解决也许更合适。

决策树算法既可以用来分类也可以用来回归，在本书中，我们会使用决策树（*https://en.wikipedia.org/wiki/Decision_tree_learning*）算法对航班延误进行回归与分类。

特征

顾名思义，特征就是样本的一个特征。在软件术语中，如果样本是对象，特征就是对象的字段或属性。对于统计预测而言，两个以上的特征组成了训练数据。至少需要两个特征，因为至少需要一个字段作为预测模型的输出，而还需要一个字段作为预测的依据。

有时候特征已经是训练数据的一部分，被放在单独的字段中。有时候，我们需要进行特征工程来从数据中提取训练模型要用的数据。

我们要用的模型会使用决策树算法。因为以下几个原因，决策树很重要。第一，决策树既能用来分类也能用来回归。而且在分类和回归之间的转换也只需一行代码就能实现。第二，用决策树可以确定并共享给定训练集的特征重要性（*feature importance*）。

特征重要性表示训练数据中哪些特征对于创建精准的模型更重要。这是无价的，能让我们找到特征工程的重点及改进预测结果的方式。而且通过表示哪些特征和要预测特征有关系，这也使我们能洞察底层数据。

回归

最简单的回归分析是线性回归。Stat Trek 将线性回归定义如下：

> 在因果关系中，自变量是因，因变量是果。最小二乘法是根据自变量 X 预测因变量 Y 的求线性回归的方法。

线性回归就是找出趋势线。我们在 Excel 中都见过（如果没有，请参阅北卡罗来纳州立大学的 Excel 回归辅导）。给定一组变量来标定航班，可以用线性回归预测航班延误或提前到达的分钟数。

分类

解决问题的一种方法是定义一系列类别，把航班分类到这些类中。航班延误是连续分布的，因此延误时间不是天生适合分类的。这里的技巧是通过定义类别，把航班延误的连续分布简化为两个或更多的类别。比如，我们可以划定类似于我们将要对天气延误分布进行分桶（0 ~ 15、15 ~ 60、60+）的类别，然后根据这三个类别进行分类。

探索航班延误

本章的主题是航班延误。要预测特征，就必须先理解特征。我们先打好基础，在应用中创建延误实体，把延误具象化出来。

我们首先探索一下问题的严重性。多少航班会晚点？似乎“所有”航班都晚点，是不是？这个数据集是令人兴奋的，因为它可以回答这样的问题！如 *ch07/explore_delays.py* 所示：

```
# 读取 Parquet 格式的准点数据文件
on_time_dataframe = spark.read.parquet('data/on_time_performance.parquet')
on_time_dataframe.registerTempTable("on_time_performance")
total_flights = on_time_dataframe.count()

# 起飞延误的航班
late_departures = on_time_dataframe.filter(on_time_dataframe.DepDelayMinutes >0)
total_late_departures = late_departures.count()

# 到达延误的航班
late_arrivals = on_time_dataframe.filter(on_time_dataframe.ArrDelayMinutes >0)
total_late_arrivals = late_arrivals.count()

# 起飞延误但是到达没延误的航班
on_time_heros = on_time_dataframe.filter(
  (on_time_dataframe.DepDelayMinutes >0)
  &
  (on_time_dataframe.ArrDelayMinutes <= 0)
)
total_on_time_heros = on_time_heros.count()

# 获取延误航班的百分比，精确到小数点后一位
pct_late = round((total_late_arrivals / (total_flights * 1.0)) * 100, 1)

print("Total flights: {:,}".format(total_flights))
print("Late departures: {:,}".format(total_late_departures))
print("Late arrivals: {:,}".format(total_late_arrivals))
print("Recoveries: {:,}".format(total_on_time_heros))
print("Percentage Late: {}%".format(pct_late))
```

结果如下：

```
Total flights:   5,819,079
Late departures: 2,125,618
Late arrivals:   2,086,896
Recoveries:  606,902
Percentage Late: 35.9%
```

哇，延误的航班占到 35.9%！看来航班延误的问题确实不小。航班平均延误多久呢？

```
# 获取起飞和降落的平均延误分钟数
spark.sql("""
SELECT
  ROUND(AVG(DepDelay),1) AS AvgDepDelay,
  ROUND(AVG(ArrDelay),1) AS AvgArrDelay
FROM on_time_performance
"""
).show()
+-----------+-----------+
|AvgDepDelay|AvgArrDelay|
+-----------+-----------+
|9.4|4.4|
+-----------+-----------+
```

航班平均起飞晚点 9.4 分钟，到达晚点 4.4 分钟。为什么总是这么拖拉？是不是航空公司太差（所以我们经常“怒喷”它们），还是天气问题？天气不受人为控制，因此航空公司可以免于责难。我们是该生航空公司的气还是生天神的气？（就我个人而言，我害怕宙斯！）

让我们看一看部分延误的航班，特别是表明延误类别的字段。我们想要确保采样是随机的，因此我们使用 Spark 的 `DataFrame.sample` 函数获取采样数据。在编写本书初稿的时候我没有使用随机采样，所以误以为天气问题导致延误（Weather Delay）最常见，事实上天气问题导致延误的情况并不多。不要偷懒，在每次调用 `.show` 函数前插入一个 `.sample(False, 0.01)`，很简单：

```
late_flights = spark.sql("""
SELECT
  FlightDate,
  ArrDelayMinutes,
  WeatherDelay,
  CarrierDelay,
  NASDelay,
  SecurityDelay,
  LateAircraftDelay
FROM
  on_time_performance
WHERE
```

```
    WeatherDelay IS NOT NULL
    OR
    CarrierDelay IS NOT NULL
    OR
    NASDelay IS NOT NULL
    OR
    SecurityDelay IS NOT NULL
    OR
    LateAircraftDelay IS NOT NULL
  ORDER BY
    FlightDate
  """)

late_flights.sample(False, 0.01).show()
```

结果如下[1]：

ArrDelayMinutes	WeatherDelay	CarrierDelay	NASDelay	SecurityDelay	LateAircraftDelay
21.0	0.0	0.0	21.0	0.0	0.0
17.0	0.0	0.0	17.0	0.0	0.0
27.0	0.0	2.0	25.0	0.0	0.0
19.0	0.0	0.0	19.0	0.0	0.0
157.0	0.0	155.0	2.0	0.0	0.0
19.0	0.0	8.0	11.0	0.0	0.0
24.0	0.0	14.0	0.0	0.0	10.0
105.0	0.0	0.0	0.0	0.0	105.0
46.0	0.0	16.0	15.0	0.0	15.0
22.0	0.0	0.0	20.0	0.0	2.0
35.0	0.0	11.0	24.0	0.0	0.0
67.0	0.0	35.0	32.0	0.0	0.0
39.0	0.0	15.0	5.0	0.0	19.0
21.0	0.0	0.0	21.0	0.0	0.0
204.0	0.0	8.0	0.0	0.0	196.0
31.0	0.0	0.0	0.0	0.0	31.0
16.0	0.0	0.0	16.0	0.0	0.0
50.0	0.0	0.0	0.0	0.0	50.0
23.0	0.0	0.0	23.0	0.0	0.0
36.0	0.0	23.0	13.0	0.0	0.0

针对不同延误类别的解释可以在联邦航空管理局（FAA）网站（*http://aspmhelp.faa.gov/index.php/Types_of_Delay*）上找到。

这样一个小采样告诉了我们什么？由于航空公司的问题而导致延误很常见，有时甚至是很严重的延误。NAS 延误（NASDelay；因国家空域系统（NAS）的航空流量管制和航空管制导致的延误）和航空公司导致的延误差不多频繁。安全问题的延误很少见，而前序航班导致的延误（前序航班没能按时抵达导致顺延）则很常见，有时也有很严重的延误。

1 下表省略了第一列。——译者注

小规模采样数据是熟悉数据的良好方式，但是采样数据可能具有欺骗性。我们想要能信赖的真实结果，因此让我们来定量查询一下延误原因。各种延误原因分别占多少百分比？我们将使用到达延误进行总数计算——这是我们必须妥协的一种简化，毕竟有些延误影响起飞，有些则影响飞行时间：

```
# 计算各种延误原因所占的百分比
total_delays = spark.sql("""
SELECT
  ROUND(SUM(WeatherDelay)/SUM(ArrDelayMinutes) * 100, 1) AS pct_weather_delay,
  ROUND(SUM(CarrierDelay)/SUM(ArrDelayMinutes) * 100, 1) AS pct_carrier_delay,
  ROUND(SUM(NASDelay)/SUM(ArrDelayMinutes) * 100, 1) AS pct_nas_delay,
  ROUND(SUM(SecurityDelay)/SUM(ArrDelayMinutes) * 100, 1) AS pct_security_delay,
  ROUND(SUM(LateAircraftDelay)/SUM(ArrDelayMinutes) * 100, 1) AS
     pct_late_aircraft_delay
FROM on_time_performance
""")
total_delays.show()
```

结果如下（已根据页面调整了格式）：

pct_weather_delay	pct_carrier_delay	pct_nas_delay	pct_security_delay	pct_late_aircraft_delay
4.5	29.2	20.7	0.1	36.1

结果并不完美，各种延误原因加起来不等于 100%。这是因为我们刚才的简化忽略了延误分为起飞延误和到达延误。不管怎么说，我们还是得到了一些大体感受；采样信息是有意义的。大多数延误都是因为同一架飞机前序航班的延误而导致的，这对飞机排班中的后续航班有级联影响。对于航班本身产生的延误，大部分都是航空公司导致的。具体来说，有 29% 的延误是航空公司的原因，而由于航空流量管制导致的延误占 21%，由于天气原因导致的延误仅占 4.5%。

我们先前的问题的答案已经显而易见了：我们通常应该怪航空公司。不过，不是航空公司导致的所有延误都是因为航空公司犯了错误。联邦航空管理局网站（*http://aspmhelp.faa.gov/index.php/Types_of_Delay*）解释道：

> 可能被认定为航空公司延误的情况有：飞机清洁、飞机损坏、等待转机乘客或机组人员、行李问题、飞机撞鸟、装载货物、餐饮、电脑问题、航空公司设备故障、机组人员法务问题（飞行员或空服人员滞留）、危险物品损害、工程检查、加油、照顾残疾乘客、机组人员迟到、卫生间清扫、维护、机票超卖、瓶装水服务、闹事乘客驱逐、登机入座或装填随身行李过慢、机身配重平衡等。

换句话说，有的时候尽管航空公司没有犯错，还是会有些令人恶心的事发生。我们没有数据可以了解航空公司犯错误所占的比例。对于我们本章的任务“预测航班延误”来说，我

们能做的最多的就是分析每个航空公司的整体延误情况。我们没办法建模预测飞机撞鸟事故或乘客闹事事件。

现在我们已经熟悉了航班延误，那么让我们把提取的一些特征拿出来进行简单的分类和回归吧。

使用 PySpark 提取特征

要使用特征，就要先把它们从宽泛的数据集中提取出来。首先，让我们使用 PySpark 少量提取几个特征，并加上对应的延误时间。在做这件事之前，我们需要确定要预测的特征是什么。记录中有两个用分钟表示的延误字段：`ArrDelayMinutes`（到达延误分钟）和 `DepDelayMinutes`（起飞延误分钟）。我们要预测的是哪一个？

根据我们的用例，用户似乎想知道两件事：一是航班起飞是否会延误，要延误多久；二是航班到达是否会延误，要延误多久。让我们把这两个值都包含到训练数据中。至于其他要提取的特征，稍加思考就会发现有一些必然要包含进来。例如，某些机场的延误比别的机场更多，因此起飞机场和降落机场的数据毫无疑问都是有意义的。在飓风季和降雪季，航班更容易延误，因此航班在一年中的哪个月也是有意义的。某些航空公司比别的航空公司更准时。最后，某些路线比其他路线更容易延误，因此航班号也有用。

我们还要加上航班唯一标识符的最后一个要素：航班日期。航班被 `FlightDate`（航班日期）、`Carrier`（航空公司）、`FlightNum`（航班号）、`Origin`（起飞地）及 `Dest`（目的地）所唯一标识。要始终保留一条唯一标识记录的所有字段，这样调试起来也会更容易。

这就是我们一开始要用到的所有特征。用的特征越多，情况就越复杂，因此我们还是尽量简单点，一开始只要用几个特征就够了。等你完成了 `sklearn` 管道配置，能够快速迭代并判断一个特征是否有用时，就可以尝试引入更多特征了。

这些特征都是简单且呈表格状的，因此要把它们选出来并存储为 JSON 格式用于模型读取也比较简单。

让我们挑出并查看我们需要的特征。如 *ch07/extract_features.py* 所示：

```
import sys, os, re
import iso8601
import datetime

# 读取 Parquet 格式准点数据文件
on_time_dataframe = spark.read.parquet('data/on_time_performance.parquet')
on_time_dataframe.registerTempTable("on_time_performance")
```

```
# 选出一些要用的特征
simple_on_time_features = spark.sql("""
SELECT
  FlightNum,
  FlightDate,
  DayOfWeek,
  DayofMonth AS DayOfMonth,
  CONCAT(Month, '-',  DayofMonth) AS DayOfYear,
  Carrier,
  Origin,
  Dest,
  Distance,
  DepDelay,
  ArrDelay,
  CRSDepTime,
  CRSArrTime
FROM on_time_performance
""")
simple_on_time_features.show()
```

结果如下（已经为适应页面宽度裁剪过了）：

FlightNum	FlightDate	...	Carrier	Origin	Dest	Distance	DepDela	ArrDelay	CRSDepTime	CRSArrTime
1519	...-01	...	AA	DFW	MEM	432.0	-3.0	-6.0	1345	1510
1519	...-01	...	AA	MEM	DFW	432.0	-4.0	-9.0	1550	1730
2349	...-01	...	AA	ORD	DFW	802.0	0.0	26.0	1845	2115
1298	...-01	...	AA	DFW	ATL	731.0	100.0	112.0	1820	2120
1422	...-01	...	AA	DFW	HDN	769.0	78.0	78.0	0800	0925
1422	...-01	...	AA	HDN	DFW	769.0	332.0	336.0	1005	1320
2287	...-01	...	AA	JAC	DFW	1047.0	-4.0	21.0	0800	1200
1080	...-01	...	AA	EGE	ORD	1007.0	null	null	1415	1755
1080	...-01	...	AA	ORD	EGE	1007.0	null	null	1145	1335
2332	...-01	...	AA	DFW	ORD	802.0	null	null	0740	0955
194	...-01	...	AA	DFW	ATL	731.0	null	null	1150	1445
356	...-01	...	AA	ATL	DFW	731.0	-5.0	1.0	1640	1805
356	...-01	...	AA	DFW	ATL	731.0	-4.0	-11.0	1300	1600
2396	...-01	...	AA	DFW	ATL	731.0	76.0	86.0	1955	2250
1513	...-01	...	AA	ATL	DFW	731.0	-2.0	-7.0	1045	1215
1513	...-01	...	AA	DFW	ATL	731.0	-5.0	-25.0	0700	1005
937	...-01	...	AA	DFW	EGE	721.0	35.0	17.0	1600	1720
937	...-01	...	AA	EGE	LAX	748.0	10.0	-12.0	1805	1920
1212	...-01	...	AA	DFW	SDF	733.0	null	null	1145	1440
1212	...-01	...	AA	SDF	DFW	733.0	null	null	1520	1640

发现有些航班没有延误信息。让我们把那些没有延误信息的过滤掉，对数据排序，然后存储为单个 JSON 文件：

```
# 过滤 null，那些数据没用
filled_on_time_features = simple_on_time_features.filter(
  simple_on_time_features.ArrDelay.isNotNull()
  &
  simple_on_time_features.DepDelay.isNotNull()
)
```

现在，我们需要把所有的日期和时间从字符串（*https://docs.python.org/3.5/library/datetime.html*）表示形式转为数学形式，否则我们的预测算法没办法理解日期间的相互关系。我们需要一些工具函数来实现：

```
# 我们需要把时间转换为时间戳格式，而不是字符串或数字
def convert_hours(hours_minutes):
  hours = hours_minutes[:-2]
  minutes = hours_minutes[-2:]

if hours == '24':
  hours = '23'
  minutes = '59'

  time_string = "{}:{}:00Z".format(hours, minutes)
  return time_string

def compose_datetime(iso_date, time_string):
  return "{} {}".format(iso_date, time_string)

def create_iso_string(iso_date, hours_minutes):
  time_string = convert_hours(hours_minutes)
  full_datetime = compose_datetime(iso_date, time_string)
  return full_datetime

def create_datetime(iso_string):
  return iso8601.parse_date(iso_string)

def convert_datetime(iso_date, hours_minutes):
  iso_string = create_iso_string(iso_date, hours_minutes)
  dt = create_datetime(iso_string)
  return dt

def day_of_year(iso_date_string):
  dt = iso8601.parse_date(iso_date_string)
  doy = dt.timetuple().tm_yday
  return doy

def alter_feature_datetimes(row):
  flight_date = iso8601.parse_date(row['FlightDate'])
  scheduled_dep_time = convert_datetime(row['FlightDate'], row['CRSDepTime'])
  scheduled_arr_time = convert_datetime(row['FlightDate'], row['CRSArrTime'])
```

```
# 处理过夜航班
if scheduled_arr_time <scheduled_dep_time:
  scheduled_arr_time += datetime.timedelta(days=1)

doy = day_of_year(row['FlightDate'])

return {
  'FlightNum': row['FlightNum'],
  'FlightDate': flight_date,
  'DayOfWeek': int(row['DayOfWeek']),
  'DayOfMonth': int(row['DayOfMonth']),
  'DayOfYear': doy,
  'Carrier': row['Carrier'],
  'Origin': row['Origin'],
  'Dest': row['Dest'],
  'Distance': row['Distance'],
  'DepDelay': row['DepDelay'],
  'ArrDelay': row['ArrDelay'],
  'CRSDepTime': scheduled_dep_time,
  'CRSArrTime': scheduled_arr_time,
}
```

在实践中，迭代实现这些函数花了一个小时。那么使用这些函数就很简单了：

```
timestamp_features = filled_on_time_features.rdd.map(alter_feature_datetimes)
timestamp_df = timestamp_features.toDF()
```

在每次向量化处理之前，都要对数据进行显式排序。不要把排序交给系统默认完成。如果那样做的话，软件版本的改变或其他某些未知的原因都可能彻底改变训练数据的排序，进而导致结果错乱。这是灾难，会产生令人困惑的结果，要尽一切可能避免这种情况发生。因此有必要对训练数据进行显式排序，以避免排序依据的不确定性：

```
# 显式排序数据，始终保持有序
# 不要抱有侥幸心理
sorted_features = timestamp_df.sort(
  timestamp_df.DayOfYear,
  timestamp_df.Carrier,
  timestamp_df.Origin,
timestamp_df.Dest,
  timestamp_df.FlightNum,
  timestamp_df.CRSDepTime,
  timestamp_df.CRSArrTime,
)
```

把文件复制到一个 JSON 行文件中，然后查看：

```
# 存储为单个 JSON 文件并使用 bzip2 压缩
sorted_features.repartition(1).write.mode("overwrite") \
  .json("data/simple_flight_delay_features.json")
os.system("cp data/simple_flight_delay_features.json/part*
  data/simple_flight_delay_features.jsonl")
```

```
os.system("bzip2 --best data/simple_flight_delay_features.jsonl")
os.system("bzcat data/simple_flight_delay_features.jsonl.bz2 >>
  data/simple_flight_delay_features.jsonl")
```

现在查看结果：

```
$ bzcat data/simple_flight_delay_features.jsonl.bz2 | head -5
{"FlightNum":"1024",
 ...
 "Carrier":"AA",
 "Origin":"ABQ",
 "Dest":"DFW",
 "DayOfYear":"1-1",
 "Distance":569.0,
 "DepDelay":14.0,
 "ArrDelay":13.0
}
{"FlightNum":"1184",
 ...
 "Carrier":"AA",
 "Origin":"ABQ",
 "Dest":"DFW",
 "DayOfYear":"1-1",
 "Distance":569.0,
 "DepDelay":14.0,
 "ArrDelay":17.0
}
{"FlightNum":"336",
 ...
 "Carrier":"AA",
 "Origin":"ABQ",
 "Dest":"DFW",
 "DayOfYear":"1-1",
 "Distance":569.0,
 "DepDelay":-2.0,
 "ArrDelay":36.0
}
{"FlightNum":"125",
 ...
 "Carrier":"AA",
 "Origin":"ATL",
 "Dest":"DFW",
 "DayOfYear":"1-1",
 "Distance":731.0,
 "DepDelay":-1.0,
 "ArrDelay":-21.0
}
{"FlightNum":"1455",
 ...
 "Carrier":"AA",
 "Origin":"ATL",
```

```
  "Dest":"DFW",
  "DayOfYear":"1-1",
  "Distance":731.0,
  "DepDelay":-4.0,
  "ArrDelay":-14.0
}
```

看起来还可以！我们的特征已经为向量化处理做好准备了。

使用 scikit-learn 构建回归模型

我们在第 3 章中说过，`scikit-learn` 是领先的适合初学者的机器学习库。它在产品级应用中也有广泛的使用。随着 Python 成为数据科学的通用语言，`sklearn` 也和 `numpy` 及 `scipy` 一同成为了数据科学的基础内容。我们要在本节中尝试使用 `sklearn` 做一个“糙快猛”的航班延误预测模型——不过，我们也会发现它的局限性。对于 Jupyter 笔记本来说，540 万条航班记录属于“大数据”。这展现出了在单机上进行科学计算的局限性，也是我们使用 Spark MLlib 的一大动机。

注意：还有一个库也被公认为是分类的利器，那就是 `xgboost`（*https://github.com/dmlc/xgboost*）。它用起来与 `sklearn` 的 `GradientBoosted` 分类器和回归器差不多，不过实现上的一些差异使得 `xgboost` 能对于我们将要介绍的这种分类任务做得比其他任何工具都要好。这个库的作者是数据科学大神 Hadley Wickham（*http://hadley.nz/*），支持 Python、R、Java、Scala 及 C++。我们先介绍线性回归，然后再介绍梯度渐进回归。

读取数据

我们先来用 Python 读取数据并构建模型。参照 *ch07/train_sklearn_model.py*，或者使用 *ch07/Predicting flight delays with sklearn.ipynb* 提供的 Jupyter 笔记本跟着做。注意在 Jupyter 笔记本中，我们需要把数据规模通过采样从 540 万条记录降到 100 万条记录。要在本地执行 Jupyter 笔记本，我们只要在项目根目录下运行 `jupyter notebook`，然后打开 *ch07/Predicting flight delays with sklearn*。

我们的特征文件有 1.6GB，因此加载时间可能需要 1 分钟。如果你是直接把代码粘贴到 iPython 中执行，那么最好用一个新的 iPython 窗口，而不是使用运行 PySpark 时用过的那个窗口。否则，你可能会内存不足。你可能还要关掉一些用不到的程序，让系统有充足的内存来运行示例代码。

如果超过 10 分钟系统都没有响应，就用 Ctrl+C 组合键来杀掉进程，按照下一节中的说明减小处理的数据规模来解决这个问题。还要注意的是如果 Jupyter 笔记本有问题，则可以尝试在普通的 iPython 控制台中执行代码，看看能不能成功。探索和发现单机 Python 的极限也是本节的要点之一，不过如果有 16GB 的内存，这些代码应该是能跑通的。

现在，让我们开始读取特征吧！

```
import sys, os, re
sys.path.append("lib")
import utils

import numpy as np
import sklearn
import iso8601
import datetime
print("Imports loaded...")

# 读取并检查训练数据的大小。可能要花一分钟时间。
training_data = utils.read_json_lines_file(
  data/simple_flight_delay_features.jsonl'
)
print("Training items: {:,}".format(len(training_data))) # 5,714,008
print("Data loaded...")
```

注意：训练数据的规模挺大，这会达到我们本地机器的极限（假设我们的内存不超过 16GB）。在迁移到 Spark MLlib 之前，我们会找找有没有什么别的办法能解决这一问题。

让我们看一看数据占多少字节，同时取一条记录出来看看：

```
# 在修改记录之前先查看一条
print("Size of training data: {:,} Bytes".format(sys.getsizeof(training_data)))
print(training_data[0])
```

结果如下：

```
Size of training data: 50,897,424 Bytes

{'ArrDelay': 13.0, 'DepDelay': 14.0, 'DayOfYear': '1-1', 'FlightNum': '1024',
'FlightDate': '2015-01-01', 'Distance':569.0,
'Carrier': 'AA', 'Origin': 'ABQ', 'Dest': 'DFW'}
```

数据采样

现在，我们到了一个分岔路口。如果你在使用 iPython 控制台执行代码，和在其他章节中一样，那么你处理全部的 540 万条记录不会遇到什么问题。然而，如果你在用 Jupyter 笔记本，那么这个量级的记录可能会压垮你的机器。尽管我的新款 MacBook Pro 有 16 GB 内存，我还是遇到了问题，因此我需要对数据进行采样。在 Jupyter 笔记本中，运行如下代码通过采样来把 540 万条记录减少到 100 万条：

```
# 在不使用 iPython 的情况下，采样出 100 万条记录
training_data = np.random.choice(training_data, 1000000)
print("Sampled items: {:,}".format(training_data))
print("Data sampled...")
```

输出如下：

```
Sampled items: 1,000,000
Data sampled...
```

如果你在使用 iPython 控制台，则可以跳过这段代码，除非你在本节后续的代码执行中遇到了问题。

本章后续的代码输出中所包含的统计信息都是使用 iPython 控制台基于全部 540 万条记录得到的。如果仅使用了 100 万条记录，则那些统计的规模大约为书中的 20%。

向量化处理结果

接下来我们需要提取并向量化处理结果集（我们尝试预测的值）：航班到达延误。对于分类和回归两种算法而言，特征和结果的编码方式有所不同，因此我们现在要决定：选择分类还是回归？如果要对航班延误进行分类，我们需要把延误时间的分钟数的数值映射到类似“准点”“晚点”“严重晚点”这些分类中。而如果要进行回归分析，则只需要直接使用延误时间的数值。因为后者要简单一些，所以我们就先选择回归。

延误有两种类型，让我们先从到达延误开始。我们要从训练数据中提取到达延误时间字段，转为 numpy.array 格式的数组。这样我们就向量化（*vectorize*）处理了数据，将其转化成了用于训练回归分析的 y 值：

```
# 把我们的结果从数据中剥离出来，向量化处理并计算大小
results = [record['ArrDelay'] for record in training_data]
results_vector = np.array(results)
print("Results vectorized size: {:,} Bytes".format(sys.getsizeof(
   results_vector)))
print("Results vectorized...")
```

结果如下：

```
Results vectorized size: 45,712,160 Bytes
Results vectorized...
```

矩阵是多维数组，而 numpy.array 是矩阵的一种高效表示。由于这里数据的值已经是浮点数形式了，因此我们不需要额外做特征提取。向量可以直接进行数学计算而无须借助低效的循环。向量化处理后还可以利用显卡（GPU）加速一些数学计算。向量化处理后，我们的结果数据大小仅为 45MB。

仅供非商业用途或交流学习使用

准备训练数据

现在需要对训练数据的特征进行编码。首先删掉到达延误字段，因为这属于结果而不是训练数据。然而我们还是在预测到达延误的训练数据中保留了出发延误时间。这就是说在进行预测时，出发延误也是特征之一。

我们也不需要航班日期，因为在预测未来的航班时，这些日期在历史数据中不会出现。当然，我们可以使用日期的一些特征，比如星期几、一个月的第几天、一年中的第几天，因为直觉告诉我们例如圣诞节那天的航班会比普通日子更容易延误：

```
# 从训练数据中删掉两个延误字段和航班日期字段
for item in training_data:
  item.pop('ArrDelay', None)
  item.pop('FlightDate', None)
print("ArrDelay and FlightDate removed from training data...")
```

接下来，我们要把日期/时间字段转为 UNIX 时间（UNIX 时间是指以格林尼治标准时间的 1970 年 1 月 1 日 0 点为基准的秒数）。这样我们的回归分析就能用数字的方式理解时间了，这也是程序理解任何事物的唯一方式（即使是类似“出发城市 ATL”这样的特征也是一样，我们稍后会讲解）：

```
# 必须把时间字符串转为 UNIX 时间
for item in training_data:
  if isinstance(item['CRSArrTime'], str):
    dt = iso8601.parse_date(item['CRSArrTime'])
    unix_time = int(dt.timestamp())
    item['CRSArrTime'] = unix_time
  if isinstance(item['CRSDepTime'], str):
    dt = iso8601.parse_date(item['CRSDepTime'])
    unix_time = int(dt.timestamp())
    item['CRSDepTime'] = unix_time
print("CRSArr/DepTime converted to unix time...")
```

向量化处理特征

现在，我们需要将特征进行编码和向量化处理。在这一过程中，数量（也就是连续（*continuous*）值）直接以数量的形式保留。然而，我们的很多特征是称名（*nominal*）数据，代表分类而不是数值。也就是说，它们是事物的称名（比如 ATL 表示一个机场）而不是数量。统计推断是基于向量化的数值数据进行的，因此我们需要把分类转为数量，并构建成矩阵提供给回归分析模型。如果还是不明白，你可以看看 Laerd Statistics 提供的数据类型指南。

仅供非商业用途或交流学习使用

幸运的是，我们有 sklearn 的支持：sklearn 的 DictVectorizer（*http://bit.ly/2omLxPB*）类使用哈希（*http://bit.ly/2pTcI4U*）来把 Python 的 dict 直接转为特征向量（稀疏矩阵）。（也有其他一些更高级的编码称名特征的方式。）这可能会花费一些时间来执行，电脑腕托部位也可能会发热，因为电脑要进行大量计算：

```
# 使用 DictVectorizer 把特征 dict 转化为向量
from sklearn.feature_extraction import DictVectorizer

print("Original dimensions: [{:,}]".format(len(training_data)))
vectorizer = DictVectorizer()
training_vectors = vectorizer.fit_transform(training_data)
print("Size of DictVectorized vectors: {:,} Bytes".format(
  training_vectors.data.nbytes))
print("Training data vectorized...")
```

结果如下：

```
Original dimensions: [5,714,008]
Size of DictVectorized vectors: 500,205,168 Bytes
Training data vectorized...
```

fit_transform 到底是如何向量化处理数据的？它结合了 fit 和 transform 方法，这两个方法本来也经常一起使用。首先 fit 创建出一组下标，把称名数据映射到下标对应的矩阵列。然后，transform 根据刚才的矩阵下标为每个特征分配一列，而每个样本则组成矩阵的一行。这样 dict 就被转为了 numpy.array。我们的数据就这样从分类转为了矩阵，矩阵中的一列表示训练集中的各条记录是否存在该列对应的分类（见图 7-2）。

仅供非商业用途或交流学习使用

向量化处理一条记录中的字符串

```
{
 'AirTime': 93.0,
 'ArrTime': 1420153920000000000,
 'CRSArrTime': 1420147200000000000,
 'CRSDepTime': 1420136400000000000,
 'Cancelled': 0.0,
 'Carrier': 'AA',
 'CarrierDelay': 19.0,
 'DepDelay': 100.0,
 'DepTime': 1420142400000000000,
 'Dest': 'ATL',
 ...
}
```

	AirTime	ArrTime	CRSArrTime	...	Carrier AA	Carrier DL	...	Dest ATL	Dest SFO	...
0	93.0	1420...	1420...		1	0		1	0	
1										

图 7-2　使用哈希向量化处理记录

稀疏矩阵与稠密矩阵

我们的训练数据经向量化处理后大约为 500MB，约为 1.6GB 原始 JSON 文件大小的 35%，但是为内存中 Python 变量大小的 10 倍左右。当数据中称名（类别）变量很多时，在向量化处理后数据大小会变大。不过要注意 `DictVectorizer` 返回的是稀疏矩阵而不是稠密矩阵。`DictVectorizer` 具体使用的矩阵类是 `scipy.sparse.csr_matrix`，它是一个“压缩的行稀疏矩阵”。

用稀疏矩阵编码称名数据时，空的元素不会以 0 编码，只有满的元素会以 1 编码。这种方式可以在数据稀疏时节省很多空间，比如这里的从 ATL 起飞的航班这种稀疏数据。这样，本节后面的代码才能运行在 MacBook Pro 上。而在单机上进行稠密矩阵的数学计算经常是无法实现的。

仅供非商业用途或交流学习使用

准备实验

在训练模型之前，我们要准备用来测试模型准确率的实验。我们要使用交叉验证法实现。交叉验证法要我们把数据分为测试集和训练集两类，先使用训练集训练模型，然后用测试集进行测试。交叉验证法可以防止过拟合（*over fitting*），也就是模型看起来准确率非常高，但是却没法很好地预测未来事件的情况。

如果没有把数据分为训练集和测试集，而是使用全部数据来训练模型，你也许可以在评估中得到很高的准确度，因为模型是依据输入的训练数据进行预测的。但这不是我们想要的。我们想要的是构建能处理前所未见的新数据的泛化统计模型，以真正实现对未来的预测。

sklearn 里面有一个可以用于交叉验证的模块 sklearn.model_selection。我们将使用 test_train_split 方法来把数据划分为训练集和测试集。测试集占总数据集的 10%，而其余 90% 作为训练数据：

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
  training_vectors,
  results_vector,
  test_size=0.1,
  random_state=43
)
print(X_train.shape, X_test.shape)
print(y_train.shape, y_test.shape)
print("Test train split performed...")
```

结果如下：

```
(900000, 7420) (100000, 7420)
(900000,) (100000,)
Test train split performed...
```

训练模型

这样，我们可以开始训练模型了！首先，我们使用训练数据来训练模型，然后将使用测试数据进行测试以度量模型的准确度。不妨看看 sklearn.linear_model.LinearRegression 的文档，因为大多数 sklearn 类的文档中都有用法的实例。我们首先选择线性回归算法进行尝试，因为尽可能简单的模型一般更好上手，要换其他的模型也类似（稍后我们会介绍替换算法有多简单）：

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

```
# 训练回归分析
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
print("Regressor library and metrics imported...")

regressor = LinearRegression()
print("Regressor instantiated...")

regressor.fit(X_train, y_train)
print("Regressor fit...")
```

sklearn 中有许多不同的模型可供使用，它们的用法都一样。多厉害！比如，我们可以现在换另一种算法梯度渐进树，只要引入 sklearn.emsemble.GradientBoostingRegressor，替换 sklearn.linear_model.LinearRegression 即可：

```
from sklearn.ensemble import GradientBoostingRegressor

regressor = GradientBoostingRegressor
print("Swapped gradient boosting trees for linear regression!")

# 再换回去 ...
regressor = LinearRegression()
print("Swapped back to linear regression!")
```

fit 方法接收训练数据和结果并创建出统计模型，通过推断如何根据特征预测结果，建立起从特征到结果的映射关系：

```
regressor.fit(X_train, y_train)
print("Regressor fitted...")
```

这就是 sklearn 中模型拟合的全部内容了。现在我们已经完成了模型拟合，让我们来看看模型准不准吧！

测试模型

我们先使用模型对测试数据进行预测。我们要使用回归分析器的 predict 方法实现预测，这个方法可以接收一条记录，也可以接收记录矩阵。predict 方法的使用格式和 fit 方法一样。这意味着你需要把特征从初始时的文本形式或对象形式通过向量化处理转化为矩阵形式，然后才能使用刚才拟合出的模型对真实案例进行实时预测。

我们需要把 X_test 和模型对 X_test 的预测进行比较，将结果量化并可视化。所以，我们把 X_test 传给 predict 方法：

```
predicted = regressor.predict(X_test)
print("Predictions made for X_test...")
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

获得预测结果后，我们可以使用 sklearn.metrics 中的大量度量指标。我们选择了 median_absolute_error（绝对误差中位数）和 r2_score 方法：

```
from sklearn.metrics import median_absolute_error, r2_score

medae = median_absolute_error(y_test, predicted)
print("Median absolute error:    {:.3g}".format(medae))

r2 = r2_score(y_test, predicted)
print("r2 score:                 {:.3g}".format(r2))

Median absolute error:     9.93
r2 score:                  0.829
```

文档把绝对误差中位数描述为实际结果和预测值之间差距的绝对值的中位数（越小越好，越大表示预测和实际之间的误差越大）。R2 则表示测定系数，是对新样本预测结果的一种度量。取值范围从 1 到 0，1.0 表示最好而 0.0 表示最差。这里我们的绝对误差中位数是 9.93 分钟，而测定系数 R2 为 0.829（接近 1），这说明我们的模型还不错！

回想到起飞的平均延误是 9.4 分钟，而我们的绝对误差中位数比这还要大。不过也没关系。这里我们的目标不是要创建一个很厉害的模型。我们的目标是把从特征到统计模型之间的路走通，然后进行预测并测试预测。记住，我们只选取了我们最先想到的几个特征来训练模型。缺少特征没办法做得更好！我们会在第 9 章中对模型进行改进。就目前而言，你应当学会了如何构建预测的框架。在第 9 章中，我会介绍如何迭代改进模型。

最后，我们要画出 X_test 中的到达延误时间和我们为 X_test 预测出的到达延误时间的对比图。换句话说，我们想用可视化的方式展现预测的准确性，把真实值作为 x 轴，把预测值作为 y 轴。我们可以使用 pyplot（*http://matplotlib.org/api/pyplot_api.html*）中的 pyplot.scatter 通过散点图来实现：

```
# 画图输出
import matplotlib.pyplot as plt

plt.scatter(
  y_test,
  predicted,
  color='blue',
  linewidth=1
)

plt.xticks(())
plt.yticks(())
plt.show()
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

结果见图 7-3。

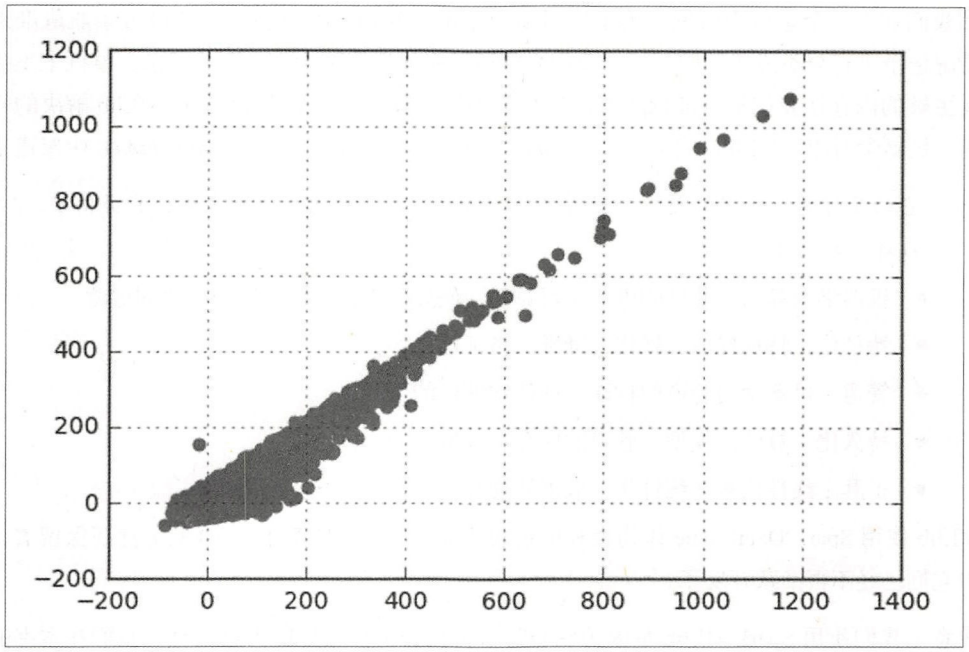


图 7-3　画出真实值和预测值的对比图

哇，有些航班晚点将近 20 个小时！图中坐标原点左下方的区域表示提前到达的航班，有一些航班早到了超过一小时！这是一张美妙的图表，斜向右上的趋势线表明我们的模型效果很好。

小结

注意：我们要通过数据采样才能用 `scikit-learn` 跑出结果。我们丢掉了 80% 的数据……不过有了 Spark，我们可能就压根不再需要采样了！我们可以使用能水平伸展到许多台机器的系统及全部的数据来做出更好的预测。

现在我们已经用 `sklearn` 构建了模型，并定量测定并画出了模型的准确性，接下来我们要使用 Spark MLlib 来构建一个分类器。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

使用 Spark MLlib 构建分类器

如我们在上一个示例中所见，为了使用 `sklearn` 对 2015 年可用的全部 540 万条航班准点情况记录进行分类或者回归分析，我们必须通过采样选出至少 100 万条记录。单机上不会有足够的内存让我们把全部训练数据都拿来训练模型。这就是要用 Spark MLlib 解决的问题。机器学习库（MLlib）指南（*http://spark.apache.org/docs/latest/ml-guide.html*）中写道：

> 它的目标是让实用机器学习变得可伸缩并且容易。从宏观角度来看，它提供了这样一些工具。
>
> - 机器学习算法：常见的机器学习算法，例如分类、回归、聚类、协同过滤。
> - 特征化：特征提取、转化、降维、特征选择。
> - 管道：机器学习管道的构建、评估和调优的工具。
> - 持久化：算法、模型、管道的保存和读取。
> - 工具：线性代数、统计学、数据处理等。

MLlib 使用 Spark DataFrame 作为表和记录的基础。尽管一些基于 RDD 的方法还保留着，但它们已经不再活跃开发了。

注意：我们使用 Spark MLlib 是因为它能跨多台机器工作来处理大量数据。我们在本书的示例中只是用了一台机器，但是不管集群规模如何，代码和过程都是一样的。通过学习使用 Spark MLlib 在单机上构建预测模型，你能学会操作 1000 台机器的集群。类似亚马逊 Elastic MapReduce（*https://aws.amazon.com/emr/*）的服务让启动能用的 Spark 集群变得只需鼠标单击即可实现。我们在本书第 1 版中讲了在云上做分析，在这一版中删除了那一章，为其他内容腾出了空间。

现在，跟我们一起用 PySpark 和 Spark MLlib 构建分类器，如 *ch07/train_spark_mllib_model.py* 所示。

使用专用结构加载训练数据

首先我们必须把训练数据读回 Spark。在我们第一次读取数据时，Spark SQL 没能识别出时间戳和日期类型，因此我们必须指明表的结构才能继续进行（就和用 `sklearn` 模型一样，把训练数据的类型分正确对于让统计推断能理解是很重要的）：

```
#
#{
#  "ArrDelay":5.0,"CRSArrTime":"2015-12-31T03:20:00.000-08:00",
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

```
#  "CRSDepTime":"2015-12-31T03:05:00.000-08:00",
#  "Carrier":"WN","DayOfMonth":31,"DayOfWeek":4,
#  "DayOfYear":365,"DepDelay":14.0,"Dest":"SAN",
#  "Distance":368.0, "FlightDate":"2015-12-30T16:00:00.000-08:00",
#  "FlightNum":"6109","Origin":"TUS"
#}
#

from pyspark.sql.types import StringType,
IntegerType, FloatType, DateType, TimestampType
from pyspark.sql.types import StructType, StructField

schema = StructType([
  StructField("ArrDelay", FloatType(), True),       # "ArrDelay":5.0
  StructField("CRSArrTime", TimestampType(), True), # "CRSArrTime":"2015-12..."
  StructField("CRSDepTime", TimestampType(), True), # "CRSDepTime":"2015-12..."
  StructField("Carrier", StringType(), True),       # "Carrier":"WN"
  StructField("DayOfMonth", IntegerType(), True),   # "DayOfMonth":31
  StructField("DayOfWeek", IntegerType(), True),    # "DayOfWeek":4
  StructField("DayOfYear", IntegerType(), True),    # "DayOfYear":365
  StructField("DepDelay", FloatType(), True),       # "DepDelay":14.0
  StructField("Dest", StringType(), True),          # "Dest":"SAN"
  StructField("Distance", FloatType(), True),       # "Distance":368.0
  StructField("FlightDate", DateType(), True),      # "FlightDate":"2015-12..."
  StructField("FlightNum", StringType(), True),     # "FlightNum":"6109"
  StructField("Origin", StringType(), True),        # "Origin":"TUS"
])

features = spark.read.json(
  "data/simple_flight_delay_features.jsonl.bz2",
  schema=schema
)
features.first()
```

结果如下：

```
Row(
  ArrDelay=13.0,
  CRSArrTime=datetime.datetime(2015, 1, 1, 10, 10),
  CRSDepTime=datetime.datetime(2015, 1, 1, 7, 30),
  Carrier='AA',
  DayOfMonth=1,
  DayOfWeek=4,
  DayOfYear=1,
  DepDelay=14.0,
  Dest='DFW',
  Distance=569.0,
  FlightDate=datetime.date(2014, 12, 31),
  FlightNum='1024',
  Origin='ABQ'
)
```

数据读取后，我们需要为分类准备数据。

仅供非商业用途或交流学习使用

处理空值

在使用 PySpark 的 MLlib 给我们提供的工具之前，我们必须先消灭 DataFrame 中记录的字段为空值的情况。否则到我们开始使用 `pyspark.ml.features`（*http://spark.apache.org/docs/latest/ml-features.html*）中的工具时会遇到代码崩溃的。

我们只需循环遍历所有的列，调用 `pyspark.sql.Column.isNull` 检查，就可以检测出每列数据中的空值：

```
null_counts = [(column, features.where(features[column].isNull()).count()) \
  for column in features.columns]
cols_with_nulls = filter(lambda x: x[1] >0, null_counts)
print(list(cols_with_nulls))
```

如果发现了空值，我们只需要调用 `DataFrame.na.fill` 把空值填上。向 `fillna`[1] 传递以列名为键、以填充值为值的 `dict` 类型的参数，这样列名对应的列中的空值就会被该填充值所填充：

```
filled_features = features.na.fill({'column_name': 'missing_replacement_value'})
```

我们的数据集里面并没有空值，不过空值在实际应用中还是很常见的，所以还要注意这一步骤。这样在特征工程和特征向量化处理过程中可以避免一些麻烦。

用 Route（路线）替代 FlightNum（航班号）

现在我们意识到对于同一对起降城市来说，航班号是不唯一的，但是航线其实是一样的。因此，让我们添加一个列 `Route`，存储 `Origin`（出发地）、`-`、`Dest`（目的地）拼接起来的字符串，比如 `ATL-SFO`。这样就能轻松告知我们的模型哪些航线经常延误，和某些机场出港航班或进港航班经常延误的情况要区分开来。

我们需要使用 `pyspark.sql.functions` 包中的两个工具来添加 `Route` 属性。`concat` 函数可以把多个字符串拼接起来，而 `lit` 方法则用来声明拼接所需的字符串常量：

```
#
# 增加 Route 以替代 FlightNum
#
from pyspark.sql.functions import lit, concat
features_with_route = features.withColumn(
  'Route',
  concat(
    features.Origin,
    lit('-'),
```

1 fillna 是 na.fill 的别名。——译者注

仅供非商业用途或交流学习使用

```
        features.Dest
      )
    )
    features_with_route.select("Origin", "Dest", "Route").show(5)
```

执行结果如下所示：

```
+------+----+-------+
|Origin|Dest|  Route|
+------+----+-------+
|   ABQ| DFW|ABQ-DFW|
|   ABQ| DFW|ABQ-DFW|
|   ABQ| DFW|ABQ-DFW|
|   ATL| DFW|ATL-DFW|
|   ATL| DFW|ATL-DFW|
+------+----+-------+
```

注意，记录数据是可以转化为 RDD 的，只要我们想这么做。做法如下：

```
def add_route(record):
  record = record.asDict()
  record['Route'] = record['Origin'] + "-" + record['Dest']
  return record

features_with_route_rdd = features.rdd.map(add_route)
```

使用 DataFrame 是因为 DataFrame 的速度比 RDD 要快很多，尽管 API 会稍显复杂一些。

对连续变量分桶以用于分类

分类算法不能用于预测类似航班延误时间（分钟）这样的连续变量；分类预测的是两个或更多类别之间的归属。因此，为了构建针对航班延误的分类器，我们要为延误的分钟数创建一组分类。

为到达延误决定分桶

在撰写本书初稿时，我们使用的分桶和湾区创业公司 FlightCaster（创建于 2009 年，于 2011 年被 Next Jump 收购）所使用的一样：准点、轻度延误、严重延误。这些分桶对应的标准来自人们按分、小时、日的分级度量时间的固有习惯。一个小时是轻度延误的直观上限。超过一个小时的延误就属于严重延误。实际到达时间在预期到达 15 分钟内的都可以认为是“准点”。如果你的案例中没有这种天然的分桶，你应该仔细分析连续变量的分布情况来决定如何分桶。

事实证明，这种分析对于我们的案例也是很重要的。在写作本书时，我们对 Spark ML 分类器模型的一个问题进行了调试，经过分析，我们发现应该使用另一组分桶。详情如

仅供非商业用途或交流学习使用

ch09/Debugging Prediction Problems.ipynb 的 Jupyter 笔记本所示。注意，GitHub 支持显示 Jupyter 笔记本，因此用它们来分享数据分析很好用——只需要提交到 GitHub 仓库，你就得到了共享的报表。笔记本对于迭代可视化开发十分方便。

用直方图迭代实现可视化。在开始前，我们先查看航班延误的整体分布情况，这可以通过把 DataFrame 转化为 RDD，然后调用 `RDD.histogram` 来获取。`RDD.histogram` 返回两个列表：一组分桶，以及每个桶的计数。然后我们使用 `matplotlib.pyplot` 画出直方图。注意由于分桶已经进行过计数了，我们就不能使用 `pyplot.hist` 了，而要使用 `pyplot.bar`（*http://bit.ly/2pSU6ls*）来根据计算好的分桶和对应计数值画出直方图。

我们选出 `ArrDelay` 列，把 DataFrame 转化为 RDD，调用 `RDD.flatMap` 把记录转化为包含单列浮点数的 RDD，这样就收集了数据：

```
%matplotlib inline

import numpy as np
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt

# 查看整体直方图
data_tuple = features\
  .select("ArrDelay")\
  .rdd\
  .flatMap(lambda x: x)\
  .histogram([-87.0, -60, -30, -15, 0, 15, 30, 60, 120])
```

接下来，我们把 `histogram` 返回元组中的所有的长条高度和分桶定义提取出来：

```
heights = np.array(data_tuple[1])

# 分桶定义比桶数多 1
full_bins = data_tuple[0]
```

由于直方图的长条是从左边开始画的，我们先把分桶定义的最右边的一个值删掉：

```
# 长条是从左边开始画的
mid_point_bins = full_bins[:-1]
```

接下来，我们使用 Python 的列表推导式来求出定义分桶的相邻值之间的范围，也就是画图时长条的宽度。我们已经决定用各分桶对应的数据范围作为长条的宽度：

```
# 长条的宽度为长条对应的数据范围
widths = [abs(i - j) for i, j in zip(full_bins[:-1], full_bins[1:])]
```

最后，我们画出直方图，指明各长条的宽度（从左端点开始），并把长条填为蓝色：

```
# 现在可以把这些长条好好画出来了
bar = plt.bar(mid_point_bins, heights, width=widths, color='b')
```

我们可以把前面这些操作总结到函数 create_hist 中，这样我们就可以复用这个函数来画出其他类似的直方图：

```
def create_hist(rdd_histogram_data):
  """ 给定 RDD.histogram 的返回，用 pyplot 画出直方图 """
  heights = np.array(rdd_histogram_data[1])
  full_bins = rdd_histogram_data[0]
  mid_point_bins = full_bins[:-1]
  widths = [abs(i - j) for i, j in zip(full_bins[:-1], full_bins[1:])]
  bar = plt.bar(mid_point_bins, heights, width=widths, color='b')
  return bar
```

得到的图表很能说明问题，这种分桶下航班稍早到达的最多（见图 7-4）。分布情况相对正常，右边稍有倾斜。现在我们要问自己一个问题：有了人们的时间观和航班延误的分布情况，以及我们要为航班延误定义分桶方式的需求，到底应该选择怎样的分桶方式呢？

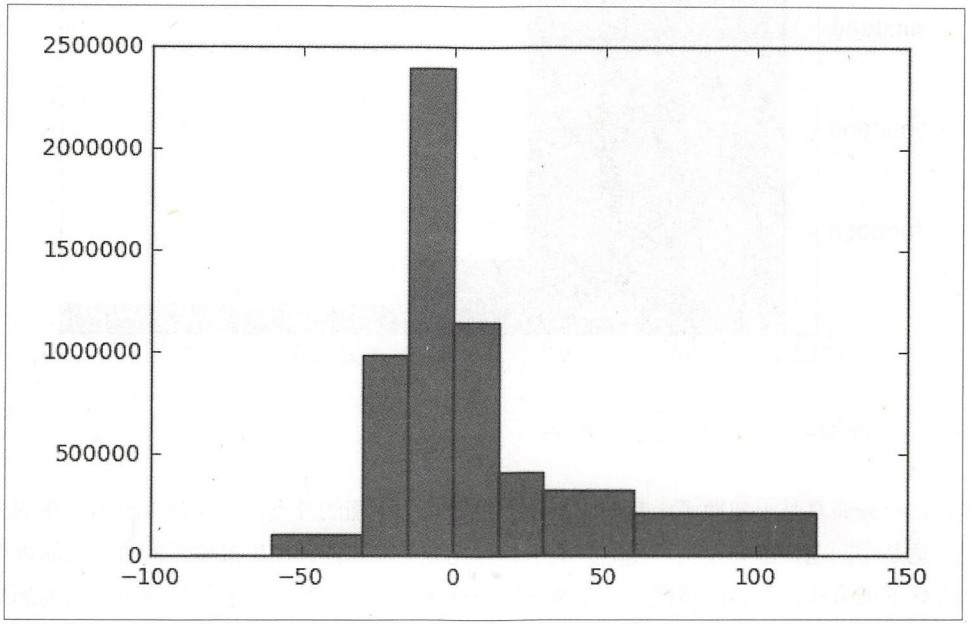


图 7-4　航班延误整体分布情况

首先，把我们考虑的第一组分桶方式可视化出来：-87 到 15，15 到 60，60 到 200。注意这种分桶方式中的第一个值 -87 是取自我们数据集中延误时间的最小值。我们选择 200 是

为了防止图表扭曲失真，尽管实际数据集中延误时间的最大值是 1971 分钟：

```
%matplotlib inline

buckets = [-87.0, 15, 60, 200]
rdd_histogram_data = features\
  .select("ArrDelay")\
  .rdd\
  .flatMap(lambda x: x)\
  .histogram(buckets)

create_hist(rdd_histogram_data)
```

结果见图 7-5。

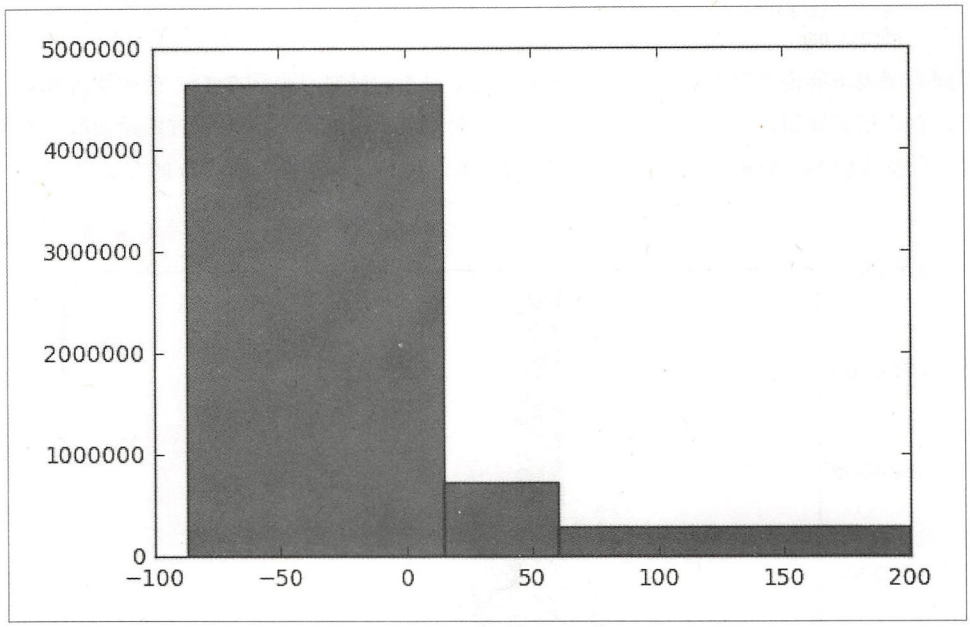


图 7-5　使用第一种分桶结构的到达延误分布

哇，这个分布真是太扭曲了。我们想创建一组理想的平衡的分类，实际是一组不平衡的分类。这是个问题，因为在不平衡的分类下，可能分类器只能预测出最常见的值，还能表现出良好的准确率。最好的情况下，分类应该让分类器模型不那么容易滥竽充数，因为那样不会有好的准确率。我们要重新思考我们的分界值。

让我们用 -87.0, -30, -15, 0, 15, 30, 60, 120 这样一组分桶来试着看看更小粒度下的分布情况[1]：

```
%matplotlib inline

buckets = [-87.0, -30, -15, 0, 15, 30, 60, 120]
rdd_histogram_data = features\
  .select("ArrDelay")\
  .rdd\
  .flatMap(lambda x: x)\
  .histogram(buckets)

create_hist(rdd_histogram_data)
```

结果见图 7-6。

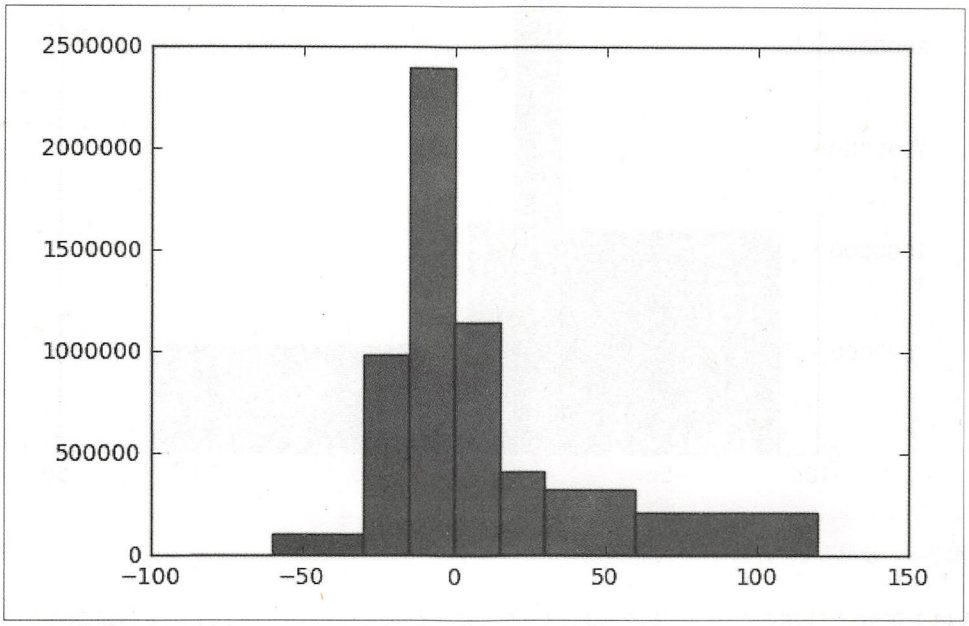


图 7-6　使用第二种分桶结构的到达延误分布

唔……看起来好一点了，不过最左边的桶和最右边的桶的频次看起来还是小了一些。我们把从 -87 到 -30 的桶和从 -30 到 -15 的桶合并起来[2]再试试：

1 此处和下面代码中的 60 为译者根据图 7-6 所加。——译者注

2 这里也把第二次尝试中 30 到 60 的桶和 60 到 120 的桶合并起来了。——译者注

仅供非商业用途或交流学习使用

```
%matplotlib inline

buckets = [-87.0, -15, 0, 15, 30, 120]
rdd_histogram_data = features\
  .select("ArrDelay")\
  .rdd\
  .flatMap(lambda x: x)\
  .histogram(buckets)

create_hist(rdd_histogram_data)
```

结果见图 7-7。

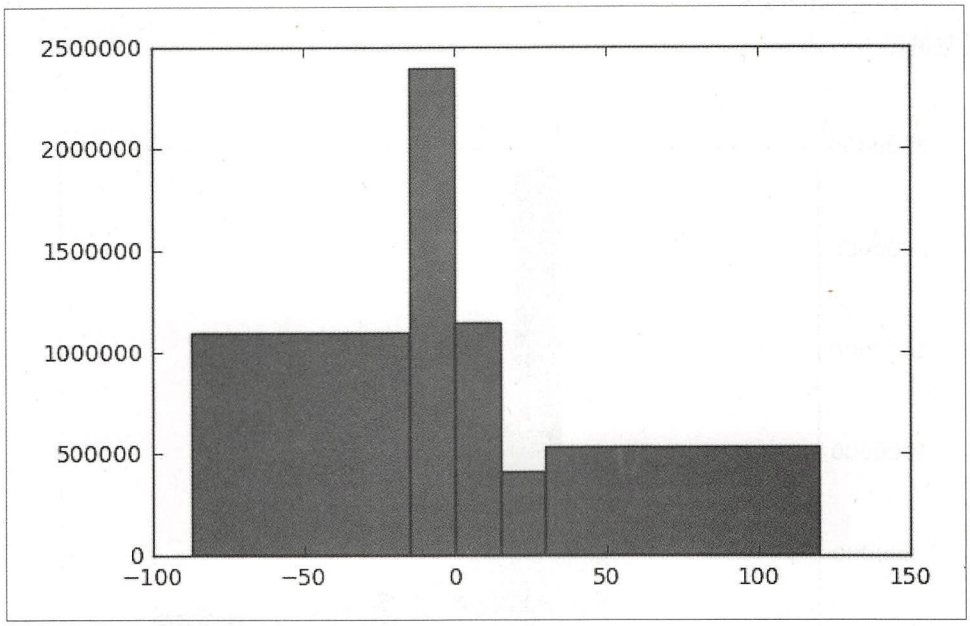


图 7-7　使用第三种分桶结构的到达延误分布

看起来更好了！然而，15 ～ 30 分钟这个桶的频次好像太小了。让我们把这个桶和 0 ～ 15 分钟的桶合并，然后重新画图：

```
%matplotlib inline

buckets = [-87.0, -15, 0, 30, 120]
rdd_histogram_data = features\
  .select("ArrDelay")\
  .rdd\
  .flatMap(lambda x: x)\
  .histogram(buckets)

create_hist(rdd_histogram_data)
```

仅供非商业用途或交流学习使用

第四次尝试的结果见图 7-8。

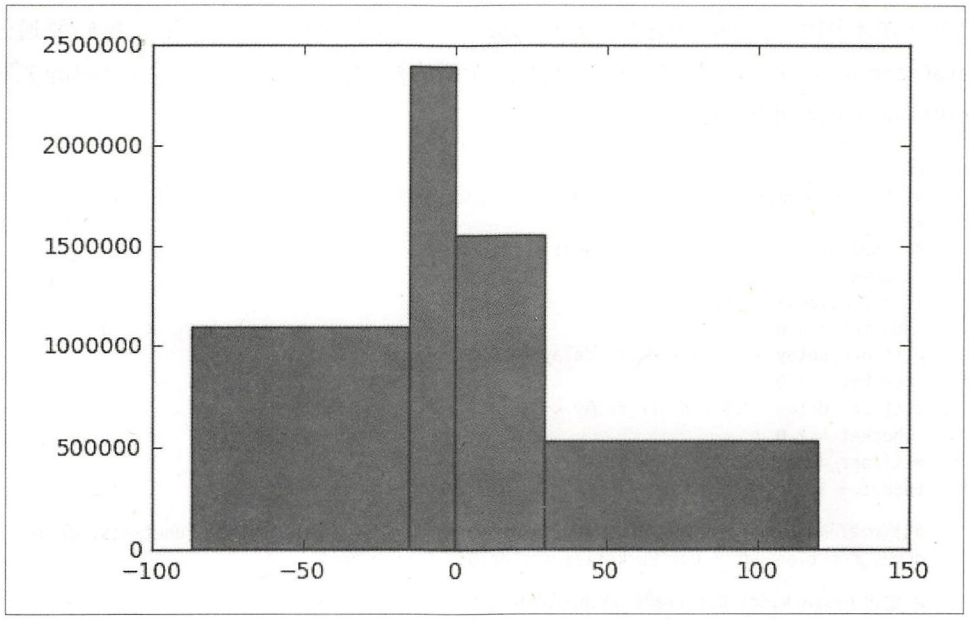


图 7-8　使用第四种分桶结构的到达延误分布

啊哈！看起来相当不错。最终的分桶是“提前很多”（提前超过 15 分钟）、“略微提前”（提前 15 分钟以内）、“略微晚点”（晚点 30 分钟以内）、“严重晚点”（晚点超过 30 分钟）。从可用性角度来讲，这种分类并不完美，但我认为它们能用。理想情况的各个桶的频次应当相等，这里已经足够接近了。

分桶探索总结。现在我们确定了合适的分桶结构，可以用来把航班延误这个连续变量转化为四个分类。注意：我们是怎样使用 PySpark 和 PyPlot 以及 Jupyter 笔记本把航班数据根据各种分桶结构进行迭代可视化的。现在这样一个笔记本就成了一份可分享的数据资产。这可以给创建该笔记本的数据科学家、产品经理以及项目工程师之间开展讨论提供一个很不错的出发点。

现在我们有了分桶的依据，让我们应用到实际数据上进行预测吧！

用 DataFrame UDF 进行分桶

我们要对数据进行分桶有两种方法：使用 DataFrame UDF 或者使用 `pyspark.ml.feature.Bucketizer`。

仅供非商业用途或交流学习使用

首先，让我们使用 UDF 根据前一节中得出的分桶结构对数据进行划分。我们要写一个函数 bucketize_arr_delay 来实现“分桶”，然后把它和 StructField 的返回类型信息（在本例中为字符串类型 StringType）一起包在 UDF 中。接下来，我们要通过 DataFrame.withColumn 应用 UDF，创建出一个新的列。最后，我们要选出 ArrDelay 列和 ArrDelayBucket 列查看对比情况：

```
#
# 使用 DataFrame UDF 对到达延误字段进行分类或“分桶”
#
def bucketize_arr_delay(arr_delay):
  bucket = None
if arr_delay <= -15.0:
  bucket = 0.0
elif arr_delay > -15.0 and arr_delay <= 0.0:
  bucket = 1.0
elif arr_delay >0.0 and arr_delay <= 30.0:
  bucket = 2.0
elif arr_delay >30.0:
bucket = 3.0 return bucket

# 把函数和 pyspark.sql.types.StructField 格式信息一起包在 pyspark.sql.functions.udf 中
dummy_function_udf = udf(bucketize_arr_delay, StringType())

# 通过 pyspark.sql.DataFrame.withColumn 添加一个分类列
manual_bucketized_features = features_with_route.withColumn(
  "ArrDelayBucket",
  dummy_function_udf(features['ArrDelay'])
)

manual_bucketized_features.select("ArrDelay", "ArrDelayBucket").show()
```

这段代码的输出如下：

```
+--------+--------------+
|ArrDelay|ArrDelayBucket|
+--------+--------------+
|    13.0|           2.0|
|    17.0|           2.0|
|    36.0|           3.0|
|   -21.0|           0.0|
|   -14.0|           1.0|
|    16.0|           2.0|
|    -7.0|           1.0|
|    13.0|           2.0|
|    25.0|           2.0|
+--------+--------------+
```

可以看到，ArrDelay 已经按我们的要求映射到了 ArrDelayBucket 中。

仅供非商业用途或交流学习使用

用 pyspark.ml.feature.Bucketizer 进行分桶

使用 Bucketizer 可以更简单地为分类模型创建分桶。我们只要把我们的划分界限定义在一个列表中，实例化出 Bucketizer，然后对特征 DataFrame 进行转化。转化的输入为 ArrDelay 字段：

```
#
# 使用 pyspark.ml.feature.Bucketizer 对 ArrDelay 进行分桶
#
from pyspark.ml.feature import Bucketizer

splits = [-float("inf"), -15.0, 0, 30.0, float("inf")]
bucketizer = Bucketizer(
  splits=splits,
  inputCol="ArrDelay",
  outputCol="ArrDelayBucket"
)
ml_bucketized_features = bucketizer.transform(features_with_route)

# 查看输出的分桶
ml_bucketized_features.select("ArrDelay", "ArrDelayBucket").show()
```

结果如下：

```
+--------+--------------+
|ArrDelay|ArrDelayBucket|
+--------+--------------+
|    13.0|           2.0|
|    17.0|           2.0|
|    36.0|           3.0|
|   -21.0|           0.0|
|   -14.0|           1.0|
|    16.0|           2.0|
|    -7.0|           1.0|
|    13.0|           2.0|
|    25.0|           2.0|
+--------+--------------+
```

可以看出结果和我们用 UDF 实现的分桶是一样的。现在我们已经创建好了 ArrDelayBucket 字段，下面就是使用 pyspark.ml.feature 中的工具来向量化处理特征了。

使用 pyspark.ml.feature 向量化处理特征

Spark MLlib 有用于多种机器学习任务的极为丰富的函数库，所以在使用 MLlib 的时候在浏览器标签页中分别打开 API 文档和 DataFrame API 文档是很有帮助的。尽管 MLlib 也有基于 RDD 的 API，但我们将使用基于 DataFrame 的 MLlib 接口。

仅供非商业用途或交流学习使用

用 Spark ML 向量化处理类别列

打开 pyspark.ml.feature 文档（*http://spark.apache.org/docs/latest/ml-features.html*），跟着这节内容边学边做。首先我们需要从 pyspark.ml.feature 中引入工具：

```
from pyspark.ml.feature import StringIndexer, VectorAssembler
```

然后对给定的称名列或类别字符串列进行去重，把去重后的所有值索引到由二进制值组成的一组向量上。要对所有的类别列（不论是字符串还是数字）进行这种操作，我们需要：

（1）配置并创建 StringIndexer 来把一列数据索引到每个不同的值对应一个数的空间上。

（2）执行 StringIndexer 的 fit 方法来获取 StringIndexer Model。

（3）在训练数据上执行 StringIndexerModel.transform，把字符串索引为一个新的列。

对每个类别变量列实现这些步骤的代码如下所示：

```
# 把类别列转化为确定的特征向量，然后删掉中间结果字段
for column in ["Carrier", "DayOfMonth", "DayOfWeek", "DayOfYear",
               "Origin", "Dest", "Route"]:
  string_indexer = StringIndexer(
    inputCol=column,
    outputCol=column + "_index"
  )
  ml_bucketized_features = string_indexer.fit(ml_bucketized_features)\
                                         .transform(ml_bucketized_features)

# 查看索引
ml_bucketized_features.show(6)
```

完成了类别型特征的索引之后，我们把这些特征和数值类型的特征组合到单个特征向量中，以供分类器使用。

用 Spark ML 向量化处理连续变量和索引值

向量化处理连续的数值类型特征不是很费劲的，因为它们已经是数值类型的了。而且现在我们有了索引值，我们的每个字符串列都有了数值表示。现在我们只需使用 VectorAssembler 就可以轻松地把数值列和索引值列组合到单个特征向量 Vector 中。然

仅供非商业用途或交流学习使用

后，删掉索引值列，因为我们已经用不到它们了：

```
# 处理连续数值和类别字段，把它们组合到单个特征向量中
numeric_columns = ["DepDelay", "Distance"]
index_columns = ["Carrier_index", "DayOfMonth_index",
                 "DayOfWeek_index", "DayOfYear_index", "Origin_index",
                 "Origin_index", "Dest_index", "Route_index"]
vector_assembler = VectorAssembler(
  inputCols=numeric_columns + index_columns,
  outputCol="Features_vec"
)
final_vectorized_features = vector_assembler.transform(ml_bucketized_features)

# 删掉索引列
for column in index_columns:
final_vectorized_features = final_vectorized_features.drop(column)

# 查看特征
final_vectorized_features.show()
```

现在我们可以开始训练分类器了！

用 Spark ML 做分类

特征已经放到单个字段 `Features_vec` 中了，接下来要设计将要进行的实验了，而实验也是创建分类器过程的一部分。我们需要一个训练数据集和一个测试数据集来进行实验。我们在之前讲过，训练数据集是用来训练模型的，而测试数据集是用来衡量准确率的。交叉验证法可以保证在实验室中创建的模型在真实数据上获得良好的表现，避免成为纸上谈兵。

用 DataFrame 分割测试 / 训练数据

和之前用 `scikit-learn` 一样，我们需要进行交叉验证。也就是说，我们要把数据划分为一个训练集和一个测试集。

DataFrame API 提供的 `DataFrame.randomSplit` 使这一切变得很容易。它接收一个表示划分比例的数组（元素之和为 1）作为参数：

```
# 划分测试 / 训练数据
training_data, test_data = final_vectorized_features.randomSplit([0.8, 0.2])
```

模型创建与拟合

随机森林分类器的引入、实例化、根据训练数据拟合，总共需要三行代码。注意：我们使用随机森林是因为它是 Spark MLlib 提供的准确率最高的支持多元分类的决策树模型。这种分类器也能提供特征重要性，在第 9 章中我们将用它来改进模型。

仅供非商业用途或交流学习使用

还要注意的是在我们第一次运行模型拟合的时候，程序抛出了异常，因为我们的单个特征去重后的总数超过了 maxBins 的默认值 32。把 maxBin 设置为异常提示中建议的值 4657，模型就成功拟合了。注意，这里需要一段时间才会执行完，先喝杯咖啡吧：

```
# 实例化并拟合随机森林分类器
from pyspark.ml.classification import RandomForestClassifier
rfc = RandomForestClassifier(
featuresCol="Features_vec", labelCol="ArrDelayBucket",
  maxBins=4657
)
model = rfc.fit(training_data)
```

接下来，我们需要评估我们创建的分类器。

模型评估

我们可以使用 MulticlassClassificationEvaluator 对分类器的表现进行评估，只要把在测试数据集上运行 pyspark.ml.classification.RandomForestClassificationModel.transform 得到的预测结果传输过来即可。虽然有几个度量值，但这里我们只看原始准确率：

```
# 使用测试数据评估模型
predictions = model.transform(test_data)

from pyspark.ml.evaluation import MulticlassClassificationEvaluator
evaluator = MulticlassClassificationEvaluator(
  labelCol="ArrDelayBucket", metricName="accuracy"
)
accuracy = evaluator.evaluate(predictions)
print("Accuracy = {}".format(accuracy))
```

结果如下：

```
Accuracy = 0.5971608857699723
```

不是特别好，但是就目前而言已经不错了。不用担心，我们会在第 9 章中让模型更为准确的。

让我们看看部分预测结果，确认是否正常。我们曾经遇到过所有的预测结果都是 0.0 的问题。查看一个包含不同预测值的采样需要动一点脑筋，因为转化时数据的顺序变了，所以我们根据预约的系统出发时间对采样进行排序，然后再展示出来：

```
# 整体检查采样数据
predictions.sample(False, 0.001, 18).orderBy("CRSDepTime").show(6)
```

仅供非商业用途或交流学习使用

结果如下（表格为适配页面宽度而裁剪过）：

ArrDelay	CRSArrTime	CRSDepTime	...	DepDelay	...	FlightDate	FlightNum	...
-10.0	...09:22:...	...05:59:	...	9.0	...	...12-31	744	...
-18.0	...13:24:...	...10:25:	...	-3.0	...	...12-31	6164	...
25.0	...03:45:...	...02:40:	...	15.0	...	...01-01	4222	...
-6.0	...08:45:...	...06:00:	...	2.0	...	...01-01	4375	...
0.0	...11:35:...	...09:35:	...	1.0	...	...01-01	4123	...
3.0	...14:45:...	...13:20:	...	2.0	...	...01-02	116	...

现在让我们查看 `Prediction` 字段的分布情况，确保没有犯同样的错误：

```
predictions.groupBy("Prediction").count().show()
```

结果如下：

Prediction	Count
0.0	3159
1.0	831281
3.0	114768
2.0	191359

“体检”通过了！

小结

通过使用 Spark，我们可以只用短短的几行代码，实现分类器或回归模型的创建、训练以及评估。令人惊奇的是 Spark 比 `scikit-learn` 还要强大。为了能用上，我们还需要部署我们的预测模型。我们会在下一章中讲解。

现在我们有了一个疑问——如何部署 Spark ML 模型呢？和 `scikit-learn` 模型不同的是，我们不能直接把它们放在网络应用中作为 API 工作，因为它们需要 Spark 平台来运行。这是下一章中会详述的内容。

本章小结

在本章中，我们已经利用我们所掌握的历史数据来预测未来了。

在下一章中，我们会深入探索这个预测模型，用它来驱动一些相关的决策。

仅供非商业用途或交流学习使用

第 8 章

部署预测系统

构建可以用来进行预测的模型是很艰苦的工作。我们需要从原始数据中提取训练数据的特征，对这些特征进行向量化处理，组合这些向量，创建一个实验，然后训练、测试并评估这个统计模型。事情很有趣，但也不轻松！

此时，我们有必要了解大多数预测系统从未离开过实验室。它们很多都没有越过当前这一步。没有人在网站上见过它们，甚至没有通过任何间接的方式感受到它们的产出。大多数预测系统在创建它们的实验室里就死掉了，很大的一个原因是创建它们的人不懂得如何部署。部署预测系统是我们本章的主题，而且出于上述原因，部署预测系统也是数据科学家成长为老手的关键技能。

本章的代码示例在 *Agile_Data_Code_2/ch08* 中。克隆代码仓库，跟上我们的脚步！

```
git clone https://github.com/rjurney/Agile_Data_Code_2.git
```

把 scikit-learn 应用部署为网络服务

把 `scikit-learn` 应用部署为网络服务相当方便。创建出模型之后，我们把它保存到磁盘上。然后我们在提供 RESTfulAPI 的网络应用启动时加载模型。

在这么做之前，我们需要先定义 API，然后从 API 回溯到模型输入的属性。我们必须把 API 的输入与模型的输入匹配起来，而 API 的参数很少会包含模型输入的全部参数。经常需要处理参数缺失的情况。

我们将使用 curl 测试我们的模型，然后把模型嵌入我们的应用程序中，提供一个能为该回归模型 API 输入参数的网页表单。当表单提交的时候，预测结果会返回。这差不多就是预测模型嵌入在真实产品中的方式了，尽管还缺一些美化和设计方面的工作。

scikit-learn 模型的保存与读取

为了实现在应用内部访问航班延误的回归模型，我们必须能在创建模型的脚本中保存模型，而且也能在提供预测 API 的网络应用中加载模型。除了预测模型，我们还要把向量化表示模型特征的对象也保存下来。

参阅 *ch07/train_sklearn_model.py*，你也可以照着 sklearn 关于持久化模型的 API 文档做。

使用 pickle 保存和加载对象

第一种保存 sklearn 模型的方法是使用 pickle（*https://docs.python.org/3/library/pickle.html*）。pickle 不是 sklearn 专用的，它是 Python 中把对象持久化到磁盘上的通用工具。使用 pickle 很简单。

我们先使用 pickle.dumps（*https://docs.python.org/3/library/pickle.html*）获取对象的字节码形式，把模型保存到磁盘上，然后使用文件句柄，把数据以二进制形式写在磁盘上（也可以通过 pickle.dump 直接实现）：

```
import pickle

project_home = os.environ["PROJECT_HOME"]

# 转存模型
regressor_path = "{}/data/sklearn_regressor.pkl".format(project_home)

regressor_bytes = pickle.dumps(regressor)
model_f = open(regressor_path, 'wb')
model_f.write(regressor_bytes)

# 转存用于向量化特征的 DictVectorizer
vectorizer_path = "{}/data/sklearn_vectorizer.pkl".format(project_home)

vectorizer_bytes = pickle.dumps(vectorizer)
vectorizer_f = open(vectorizer_path, 'wb')
vectorizer_f.write(vectorizer_bytes)
```

使用 pickle.loads 和 pickle.load 读取模型也一样简单：

```
# 读取模型
model_f = open(regressor_path, 'rb')
model_bytes = model_f.read()
regressor = pickle.loads(model_bytes)
```

```
# 读取 DictVectorizer
vectorizer_f = open(vectorizer_path, 'rb')
vectorizer_bytes = vectorizer_f.read()
vectorizer = pickle.loads(vectorizer_bytes)
```

pickle 是存储 Python 对象的一种强大而通用的方式。

使用 sklearn.externals.joblib 保存和加载模型

使用 sklearn.externals.joblib 把模型保存到磁盘上只需一行代码：

```
from sklearn.externals import joblib

# 转存模型和向量器
joblib.dump(regressor, 'data/sklearn_regressor.pkl')
joblib.dump(vectorizer, '../data/sklearn_vectorizer.pkl')
```

读取也是一样简单：

```
# 读取模型和向量器
regressor = joblib.load('../data/sklearn_regressor.pkl')
vectorizer = joblib.load('../data/sklearn_vectorizer.pkl')
```

在本章后面讲解网络应用的时候，我们会使用这个方法。

提供预测模型的准备工作

首先，显而易见的是，除了构建一个简单的 API 把预测模型部署到网络上，我们要做的事情还有很多。譬如，以训练集里的一条航班记录为例：

```
{
  "ArrDelay":5.0,
  "Carrier":"WN",
  "DayOfMonth":31,
  "DayOfWeek":4,
  "DayOfYear":365,
  "DepDelay":14.0,
  "Dest":"SAN",
  "Distance":368.0,
  "FlightNum":"6109",
  "Origin":"TUS"
}
```

除了我们要预测的 ArrDelay 字段，我们还要求出其他所有字段的值，并以向量表示，这样我们的应用就可以在 API 内部重复其行为。明白吗？但不完全是这样的。

比如，要让用户去计算并提供给定日期对应一个月、一周、一年中的第几天真的有道理吗？显然没有道理。API 参数直接接收日期才更有道理，并且对用户更加友好，这些字段的计算应该作为 API 预测过程的一部分。

那么我们就不能预期用户去了解某些字段了，比如起点和目的地之间的距离这种字段。我们需要根据我们的数据，创建出一个查询表，根据起点和目的地查出对应的值。

这些都说明了部署预测模型的一条重要原则：*如果不能实时获取数据，就不能把它纳入模型中并成功部署*。这大大限制了我们能为改进模型而做的事情。

我们的 API 的形式如下所示，这是从字段名到参数类型的映射表。通过这些值，可以求出组成航班记录训练数据的其他值：

```
api_field_type_map = \
{
  "DepDelay": float,
  "Carrier": str,
  "Date": str,
  "Dest": str,
  "FlightNum": str,
  "Origin": str
}
```

为航班延误回归分析创建 API

为了能在网络应用中提供预测，我们首先需要在应用启动时加载模型。然后，收到请求内容后，我们接受那些能根据参数求出模型需要的所有特征的请求，并把它们转化为模型要求的向量化形式。最后，我们把向量化的数据传给模型，返回 JSON 格式的结果。

让我们依次遍历各个部分。参见 *ch08/web/predict_flask.py* 和 *ch08/web/predict_utils.py* 文档。

首先，我们使用 `sklearn.externals.joblib.load` 加载模型。注意，我们引用环境变量 `$PROJECT_HOME` 来确保无论从哪个目录启动 Flask 应用程序，我们的读取工作都能正确执行：

```
# 读取回归模型
from sklearn.externals import joblib
project_home = os.environ["PROJECT_HOME"]
vectorizer = joblib.load("{}/data/sklearn_vectorizer.pkl".format(project_home))
regressor = joblib.load("{}/data/sklearn_regressor.pkl".format(project_home))
```

接下来，我们定义 API 的入口，把它定义为只接收 POST 请求，这样搜索引擎就不会触发控制器所提供的预测了，毕竟这是相对高代价的操作：

```
# 让 API 只接收 POST 请求，这样搜索引擎不会访问到它
@app.route("/flights/delays/predict/regress", methods=['POST'])
def regress_flight_delays():
```

我们使用之前定义的 api_field_type_map（从 API 字段名到参数类型的映射表）来获取前端表单传到控制器的值，把它们放到要传给回归模型的记录中：

```
api_field_type_map = \
  {
    "DepDelay": float,
    "Carrier": str,
    "Date": str,
    "Dest": str,
    "FlightNum": str,
     "Origin": str
  }
api_form_values = {}
for api_field_name, api_field_type in api_field_type_map.items():
  api_form_values[api_field_name] = request.form.get(
    api_field_name, type=api_field_type
)

# 设置直接获取的值
prediction_features = {}
prediction_features['Origin'] = api_form_values['Origin']
prediction_features['Dest'] = api_form_values['Dest']
prediction_features['FlightNum'] = api_form_values['FlightNum']
```

然后定义名为 predict_utils.get_flight_distance 的函数 API，用来求 Distance 字段，也就是出发地 Origin 和目的地 Dest 之间的距离：

```
# 设置要算的值
prediction_features['Distance'] = predict_utils.get_flight_distance(
  client, api_form_values['Origin'], api_form_values['Dest']
)
```

还要定义一个名为 predict_utils.get_regression_date_args 的函数 API，根据给定的日期参数 Date 计算对应一年中的第几天（DayOfYear）、一个月中的第几天（DayOfMonth），以及一周中的第几天（DayOfWeek）字段：

```
# 把日期转为 DayOfYear、DayOfMonth、DayOfWeek
date_features_dict = predict_utils.get_regression_date_args(
  api_form_values['Date']
)
```

```
for api_field_name, api_field_value in date_features_dict.items():
  prediction_features[api_field_name] = api_field_value
```

求出模型训练数据中一条记录的全部字段之后，我们把这条记录向量化，映射到回归模型的向量空间中：

```
# 向量化特征
feature_vectors = vectorizer.transform([prediction_features])
```

有了向量化处理好的特征，我们就可以进行预测了：

```
# 进行预测！
result = regressor.predict(feature_vectors)[0]
```

最后，我们把结果以 JSON 对象返回：

```
# 返回 JSON 对象
 result_obj = {"Delay": result}
 return json.dumps(result_obj)
```

这样，我们的回归分析 API 就完成了！

现在，为了让它跑起来，我们需要实现刚才在控制器代码中定义的 API：`predict_utils.get_flight_distance` 和 `predict_utils.get_regres sion_date_args`。

实现 predict_utils API

这意味着我们需要在 *predict_utils.py* 中实现一个名为 `get_flight_distance（ori gin,dest）` 的函数，返回两个机场之间的飞行距离。要实现这一功能，让我们使用 PySpark 在 MongoDB 中创建一张表，保存以出发地和目的地机场代码为键，以距离的英里数为值的数据，如 *ch08/origin_dest_distances.py* 所示。我们运行一条简单的 GROUP BY/AVG 查询命令来计算机场间的距离：

```
# 读取 Parquet 格式的准点数据文件
on_time_dataframe = spark.read.parquet('data/on_time_performance.parquet')
on_time_dataframe.registerTempTable("on_time_performance")
origin_dest_distances = spark.sql("""
  SELECT Origin, Dest, AVG(Distance) AS Distance
  FROM on_time_performance
  GROUP BY Origin, Dest
  ORDER BY Distance
  """)
origin_dest_distances.repartition(1).write.mode("overwrite") \
  .json("data/origin_dest_distances.json"
   )
os.system(
  "cp data/origin_dest_distances.json/part* data/origin_dest_distances.jsonl"
)
```

为了把数据载入 MongoDB 中，我们对得到的 JSON 行文件运行导入命令。如 *ch08/import_distances.sh* 所示，我们还给 `Origin/Dest` 键创建了索引：

```
# 把数据导入表中
mongoimport -d agile_data_science -c origin_dest_distances --file \
  data/origin_dest_distances.jsonl
mongo agile_data_science --eval \
  'db.origin_dest_distances.ensureIndex({Origin: 1, Dest: 1})'
```

验证数据在 MongoDB 中：

```
> db.origin_dest_distances.find({"Origin": "ATL", "Dest": "JFK"})
{
  "_id": ObjectId("583bc2e6aeb23e2f187ce737"), "Origin": "ATL",
  "Dest": "JFK",
  "Distance": 760
}
```

最后，在 *ch08/web/predict_utils.py* 中，我们使用 PyMongo 把它封装在 `predict_utils.get_flight_distance` 的函数 API 中：

```
def get_flight_distance(client, origin, dest):
  """ 获取一对机场代码之间的距离 """
  record = client.agile_data_science.origin_dest_distances.find_one({
    "Origin": origin,
    "Dest": dest,
  })
  return record["Distance"]
```

类似地，我们需要创建函数 `predict_utils.get_regression_date_args`，不过多亏了 Python 内建的 `datetime`（*https://docs.python.org/3/library/datetime.html*）库，这个函数的实现比较简单而且是纯 Python 的。`datetime` 库中有方法可以获取 `DayOfYear`、`DayOfMonth` 和 `DayOfWeek`：

```
def get_regression_date_args(iso_date):
  """ 给定 ISO 格式的日期，返回对应一年中的第几天、一个月中的第几天，
      以及一周中的第几天，因为 API 需要这些值。"""
  dt = iso8601.parse_date(iso_date)
  day_of_year = dt.timetuple().tm_yday
  day_of_month = dt.day
  day_of_week = dt.weekday()
  return {
    "DayOfYear": day_of_year,
    "DayOfMonth": day_of_month,
    "DayOfWeek": day_of_week,
  }
```

搞定了！现在我们可以对它进行测试了。

测试 API

和我们之前的做法一样，可以使用 curl（*https://en.wikipedia.org/wiki/CURL*）工具来测试航班延误回归模型 API。如 *ch08/test_regression_api.sh* 所示。通过 -XPOST 选项，curl 命令可以提交 HTTP POST 请求，并且通过 -F 参数设定表单数值：

```
#!/usr/bin/env bash

# 获取给定航班的延误预测
curl -XPOST 'http://localhost:5000/flights/delays/predict/regress' \
  -F 'DepDelay=5.0' \
  -F 'Carrier=AA' \
  -F 'Date=2016-12-23' \
  -F 'Dest=ATL' \
  -F 'FlightNum=1519' \
  -F 'Origin=SFO' \
| json_pp
```

结果如下：

```
  % Total    %  Received  %  Xferd  Average  Speed    Time    Time    Time  Current
                                    Dload   Upload   Total   Spent   Left  Speed
100   672  100     29   100    643   3394   75266 --:--:-- --:--:-- --:--:--  104k
{
   "Delay": -36.4042325748015
}
```

成功了！我们已经把一个 sklearn 预测模型部署为网络服务了。

在产品中使用 API

部署预测模型的最后一步是开发提供预测服务的页面。让我们从实现控制器开始，定义变量，form_config 用于在 Jinja 模板中生成表单：

```
@app.route("/flights/delays/predict")
def flight_delays_page():
"""Serves flight delay predictions"""

  form_config = [
    {'field': 'DepDelay', 'label': 'Departure Delay'},
    {'field': 'Carrier'},
    {'field': 'Date'},
    {'field': 'Origin'},
    {'field': 'Dest', 'label': 'Destination'},
    {'field': 'FlightNum', 'label': 'Flight Number'},
]

return render_template('flight_delays_predict.html', form_config=form_config)
```

在模板中，我们用循环遍历 form_config 来创建表单的字段。我们为结果创建一个 div，使用 jQuery 提交表单，解析并把结果放在 div 中：

```
{% extends "layout.html" %}
{% block body %}
  / <a href="/flights/delays/predict">Flight Delay Prediction</a>

  <p class="lead"style="margin: 10px; margin-left: 0px;">

    Predicting Flight Delays
  </p>

  <!-- 根据 form_config 和请求参数生成表单 -->
  <form id="flight_delay_regression"
        action="/flights/delays/predict/regress"
        method="post">
    {% for item in form_config %}
      {% if 'label' in item %}
        <label for="{{item['field']}}">{{item['label']}}</label>
      {% else %}
        <label for="{{item['field']}}">{{item['field']}}</label>
      {% endif %}
        <input name="{{item['field']}}"
               style="width: 36px; margin-right: 10px;"
               value="">
        </input>
    {% endfor %}
    <button type="submit"class="btn btn-xs btn-default"style="height: 25px">
     Submit
    </button>
  </form>

  <div style="margin-top: 10px;">
      <p>Delay: <span id="result"style="display: inline-block;"></span></p>
  </div>

  <script>
    // 为表单添加提交
    $( "#flight_delay_regression").submit(function( event ) {

      // 防止正常的表单提交
      event.preventDefault();

      // 从页面上获取一些元素的值
      var $form = $( this ),
        term = $form.find( "input[name='s']").val(),
        url = $form.attr( "action");

      // 使用 post 发送数据
      var posting = $.post( url, $( "#flight_delay_regression").serialize() );

      // 把结果放在 div 中
      posting.done(function( data ) {
        result = JSON.parse(data);
        $( "#result").empty().append( result.Delay );
```

仅供非商业用途或交流学习使用

```
        });
      });
    </script>
  {% endblock %}
```

结果很简单，也符合我们当前的要求（见图 8-1）。在这个例子里，航班提前起飞并且提前到达。

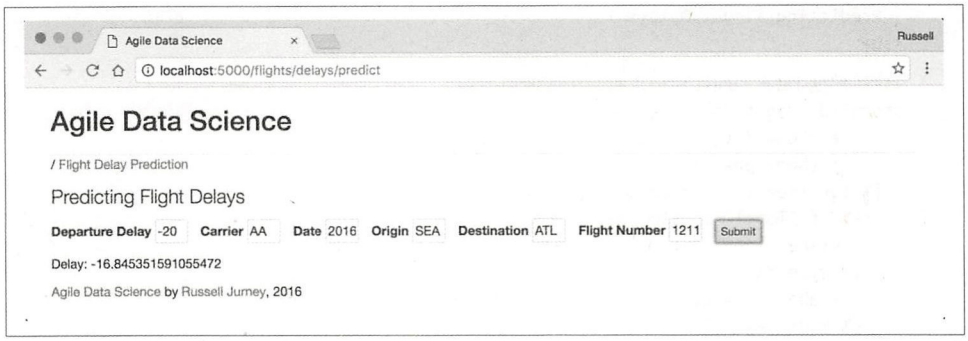


图 8-1　航班延误回归预测页面

使用 Airflow 部署批处理模式 Spark ML 应用

和部署 `scikit-learn` 应用比起来，部署 Spark ML 应用要更复杂。这是因为在我们和终端应用之间多了一层 Spark。有两种部署 Spark ML 预测模型的方式：一是定时批处理，二是通过 Spark Streaming 实现“实时”处理。两种方式我们都会介绍，先从批处理开始介绍。

为了实现数据批处理，我们需要对数据管道进行组合、调度和监控。要实现这些功能，需要使用批处理调度器，比如 `Azkaban`（*https://azkaban.github.io/*）、`Apache Oozie`（*http://oozie.apache.org/*）或者 `Apache Airflow`（*https://airflow.incubator.apache.org/*）。我们选择 Airflow，因为它正在成为主流选择，而且能与我们软件栈中的其他工具共同实现高的生产率。

我们在第 2 章中介绍过 Airflow。在本节中，将使用 Airflow 和网络应用程序一同部署数据管道，来执行使用 Spark ML 批处理进行预测所必需的端到端操作。为了简单起见，我们选择每天进行一次预测。这种频率可以适用于每天批量发送的邮件，或者构建事件摘要的推荐内容。

使用 Airflow 和 Spark 实现的批处理能够以最小约 5 分钟的粒度调度任务。如果有更高频率的需求，就要使用 Spark Streaming 了。

仅供非商业用途或交流学习使用

进行批处理模式预测的必要操作按顺序依次为：

1. 从数据中提取特征，创建训练集。
2. 使用训练数据训练分类器，保存模型以备后用。
3. 从网络应用程序收集预测请求，保存到 MongoDB 中。
4. 把 MongoDB 中的请求集中到每天的批处理对应的文件中。
5. 加载模型和今天的请求，进行实际预测，把预测结果按天进行保存。
6. 把每天的批处理的预测结果读取到 MongoDB 中。
7. 在网络应用中呈现今天的预测。

一头是训练数据，另一头是用户。我们的任务是使用在生产环境中每天运行一次的系统把两头连起来。我们先介绍各个任务的内容和构建方式，然后使用 Airflow 把任务整合起来。让我们开始吧！

在生产环境中收集训练数据

在第 7 章中我们创建了脚本 *ch07/extract_features.py* 来收集训练数据。这个脚本中的操作可以不用修改，我们只要修改输入 / 输出路径并把脚本配置为从命令行运行，就可以直接使用。因此，我把它复制到 *ch08/extract_features.py* 中，然后进行编辑。

我们把可执行的内容嵌入 `main` 函数中，这样可以通过命令行把文件系统中的基本路径作为参数传递过来。接下来我们使用命令行参数调用这个函数。让这些内容可以从命令行执行，这样 Airflow 也可以调用。启动部分和 PySpark 初始化的代码实现使得脚本既支持在开发环境中用 PySpark 解释器执行，又支持在生产环境中用命令行调用。在这个脚本中我们不需要使用日期参数，但是在其他使用每天或每小时的批量数据作为输入和输出的时候要用到日期参数。

为了让脚本能在开发环境和生产环境中运行，我们按需初始化 PySpark 环境。如果 `SparkContext` 和 `SparkSession` 已经被 PySpark 命令行初始化，我们引用变量 `sc` 和 `spark` 的时候就不会有异常抛出了，这意味着它们不会被重新初始化。这很重要，因为在命令行或 Airflow 中运行脚本时，重新初始化 `SparkContext` 会导致异常，使得进程挂掉。与此同时，如果这些变量没被初始化，脚本也会挂掉。我们的有条件初始化可以分别处理这两种运行环境：

仅供非商业用途或交流学习使用

```
#!/usr/bin/env python

import sys, os, re import json
import datetime, iso8601

# 从 Airflow 把基本路径传给 main()
def main(base_path):

  APP_NAME = "extract_features.py"

# 如果没有 SparkSession，则创建环境
try:
  sc and spark
except NameError as e:
  import findspark
  findspark.init()
  import pyspark
  import pyspark.sql

  sc = pyspark.SparkContext()
  spark = pyspark.sql.SparkSession(sc).builder.appName(APP_NAME).getOrCreate()

# 读取 Parquet 格式准点数据文件
input_path = "{}/data/on_time_performance.parquet".format(
  base_path
)
on_time_dataframe = spark.read.parquet(input_path)
on_time_dataframe.registerTempTable("on_time_performance")
```

这个脚本的工作包括把 ISO 日期字段转为日期，和我们在第 7 章中所做的一样。同样修改脚本的最后部分，在输出路径中插入基本路径。我们使用命令行参数中的日期和基本路径调用 main 函数来完成该脚本：

```
  # 保存为单个 JSON 文件
  output_path = "{}/data/simple_flight_delay_features.json".format(
    base_path
  )
  sorted_features.repartition(1).write.mode("overwrite").json(output_path)
  combine_cmd = "cp {}/part* {}/data/simple_flight_delay_features.jsonl".format(
    output_path,
    base_path
  )
os.system(combine_cmd)

if __name__ == "__main__":
main(sys.argv[1])
```

仅供非商业用途或交流学习使用

我们可以从命令行测试脚本，注意尽管没有用到日期，但是它可以使这个脚本与其他脚本保持一致[1]：

```
python ch08/extract_features.py.
```

Spark ML 模型的训练、存储与加载

和 sklearn 一样，在我们使用从 Spark ML 构建的模型进行预测之前，我们要把模型保存到硬盘上，然后加载回来。这些模型包括我们训练过的随机森林分类器，以及各种把原始数据转化为向量形式的模型。为了保存这些模型，我们来回顾一下第 7 章中创建模型的脚本，添加代码，来将整个管道中的每一个模型保存到硬盘上。这样我们就能在其他脚本中加载模型了，不用每次使用模型时都重新训练模型。这是很重要的，因为训练模型所花费的时间远大于加载模型或者使用模型进行预测所需的时间。

我们从第 7 章中复制了代码，并进行了修改，使脚本把每个模型存储为项目 *models/* 文件夹中的一个文件，如 *ch08/train_spark_mllib_model.py* 所示。用于转化训练数据的每个模型都要实现存储和加载，这样它们才能实时地处理预测请求。注意我们还把脚本修改成了能从命令行直接运行的形式，不过我们暂时不用深入探讨这部分修改。

首先，让我们保存用于到达地和出发地的分桶器。保存这种模型的代码并不复杂：

```
# 配置分桶器
splits = [-float("inf"), -15.0, 0, 30.0, float("inf")]
arrival_bucketizer = Bucketizer(
  splits=splits,
  inputCol="ArrDelay",
  outputCol="ArrDelayBucket"
)

  # 保存模型
arrival_bucketizer_path = "{}/models/arrival_bucketizer_2.0.bin".format(
  base_path
)
arrival_bucketizer.write().overwrite().save(arrival_bucketizer_path)

  # 应用模型
  ml_bucketized_features = arrival_bucketizer.transform(features_with_route)
  ml_bucketized_features.select("ArrDelay", "ArrDelayBucket").show()
```

我们要保存每个字符串字段的索引模型。幸运的是，StringIndexerModel 可以通过调用 StringIndexerModel.save 直接保存：

1 本节中作者原文一直在讲，脚本中传递了日期参数但没有使用，然而代码和最后的调用示例中都没有支持和真正地传递日期参数。——译者注

仅供非商业用途或交流学习使用

```
#
# pyspark.ml.feature 中的特征提取工具
#
from pyspark.ml.feature import StringIndexer, VectorAssembler

# 把类别字段转化为索引值
for column in ["Carrier", "DayOfMonth", "DayOfWeek", "DayOfYear",
               "Origin", "Dest", "Route"]:
string_indexer = StringIndexer(
  inputCol=column,
  outputCol=column + "_index"
)

string_indexer_model = string_indexer.fit(ml_bucketized_features)
ml_bucketized_features = string_indexer_model.transform(
  ml_bucketized_features
)

# 删掉原来的列
ml_bucketized_features = ml_bucketized_features.drop(column)

# 保存管道模型
string_indexer_output_path = "{}/models/string_indexer_model_{}.bin".format(
  base_path,
  column
)
string_indexer_model.write().overwrite().save(string_indexer_output_path)
```

还要把我们所有的数值列和索引值列整合到单个特征向量里的整合器 VectorAssembler 中：

```
# 处理连续数值和类别字段，把它们整合到单个特征向量中
numeric_columns = ["DepDelay", "Distance"]
index_columns = ["Carrier_index", "DayOfMonth_index",
                 "DayOfWeek_index", "DayOfYear_index", "Origin_index",
                 "Origin_index", "Dest_index", "Route_index"]
vector_assembler = VectorAssembler(
  inputCols=numeric_columns + index_columns,
  outputCol="Features_vec"
)
final_vectorized_features = vector_assembler.transform(ml_bucketized_features)

# 保存向量整合器
vector_assembler_path = "{}/models/numeric_vector_assembler.bin".format(
  base_path
)
vector_assembler.write().overwrite().save(vector_assembler_path)

  # 删掉原来的列
for column in index_columns:
  final_vectorized_features = final_vectorized_features.drop(column)
```

仅供非商业用途或交流学习使用

```
# 查看最终的特征
final_vectorized_features.show()
```

最后，我们训练和存储随机森林分类器本身。注意，尽管在开发时要创建一个试验并把数据集划分为训练集和测试集，但在生产环境中要使用全部数据，把测试部分的数据也一起拿来训练以获得更高准确率的做法是很常见的：

```
    # 实例化随机森林分类器并根据全部数据进行拟合
    from pyspark.ml.classification import RandomForestClassifier
    rfc = RandomForestClassifier(
      featuresCol="Features_vec",
      labelCol="ArrDelayBucket",
      predictionCol="Prediction",
      maxBins=4657,
    )
    model = rfc.fit(final_vectorized_features)

    # 保存新模型，覆盖旧模型
    model_output_path = \
      "{}/models/spark_random_forest_classifier.flight_delays.5.0.bin".format(
      base_path
    )
    model.write().overwrite().save(model_output_path)

  if __name__ == "__main__":
    main(sys.argv[1])
```

现在我们可以随时加载这些模型了，不管什么时候，也不管是 Spark 批处理模式还是 Spark Streaming。让我们尝试从命令行运行该脚本。注意，执行该脚本可能要花几分钟的时间：

```
python ch08/train_spark_mllib_model.py .
```

接下来，我们就要在网络应用中创建预测请求了。

在 MongoDB 中创建预测请求

在现实世界中，这可能对应于需要为用户每天创建预测或建议，以便将预测置于当天的用户内容中。为了把预测任务和相应的数据提供给调度器程序运行的 Spark ML 脚本，我们需要在数据库中生成表、记录需要进行的预测。网络应用程序可以轻松实现这一目标。在实际环境中，这对应那些要每天为用户进行预测或者推荐，把预测结果放入当天用户内容中的需求。

在这种情况下，网络应用会把预测请求保存为 MongoDB 表中的记录。然后，用 Airflow 调度每天的定时任务，获取当天的预测任务，传给 PySpark ML，后者会做出预测，把结果保存到另一个 MongoDB 表中。这样预测页面就可以展示最新的预测结果了。

仅供非商业用途或交流学习使用

这种工作流程比较粗糙，但是希望你能想象它所适合的场景：生成提供推荐内容的邮件、生成每日精选内容等。对于不适合使用批处理工作流的任务，我们还是使用 Spark Streaming 吧。

通过 Flask API 提供推荐任务给 MongoDB

为了在 MongoDB 中存储所需预测的记录，我们把前一节中的 Flask 网络应用改为存储预测请求而非生成相应的 `scikit-learn` 预测，如 *ch08/web/predict_flask.py* 和 *ch08/web/predit_utils.py* 所示。

注意：这是我们第一次使用 `pymongo` 插入数据。之前我们所有的控制器都只有只读操作。这一点要格外注意，因为维护可读可写的网络应用比只读的要费时很多。

该 API 的大部分代码是从我们前一节实现的 `sklearn` 回归模型 API 中复制过来的。在 *ch08/web/predict_flask.py* 中，我们使用与之前 API 一样的工具函数来补充完整通过 POST 请求提交过来的记录。另外，我们在记录中添加了一个 ISO 格式（*https://en.wikipedia.org/wiki/ISO_8601*）的时间戳 `Timestamp`，插入 MongoDB 表 `prediction_tasks` 中。最后，我们以 JSON 格式返回记录，验证请求已被正确处理：

```
# 让 API 只接受 POST 请求，这样搜索引擎不会访问到
@app.route("/flights/delays/predict/classify", methods=['POST'])
def classify_flight_delays():
  """ 分类分析航班延误的 POST API"""
  api_field_type_map = \
    {
      "DepDelay": int,
      "Carrier": str,
      "FlightDate": str,
      "Dest": str,
      "FlightNum": str,
      "Origin": str
    }

  api_form_values = {}
  for api_field_name, api_field_type in api_field_type_map.items():
  api_form_values[api_field_name] = request.form.get(
    api_field_name, type=api_field_type
  )

# 设置无须修改的值，不包括日期
prediction_features = {}
for key, value in api_form_values.items():
  prediction_features[key] = value
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

```
# 设置计算得出的值
prediction_features['Distance'] = predict_utils.get_flight_distance(
  client, api_form_values['Origin'],
  api_form_values['Dest']
)

# 把日期转为 DayOfYear, DayOfMonth, DayOfWeek
date_features_dict = predict_utils.get_regression_date_args(
  api_form_values['FlightDate']
)
for api_field_name, api_field_value in date_features_dict.items():
  prediction_features[api_field_name] = api_field_value

# 添加时间戳
  prediction_features['Timestamp'] = predict_utils.get_current_timestamp()

client.agile_data_science.prediction_tasks.insert_one(
  prediction_features
)
return json_util.dumps(prediction_features)
```

我们在 *ch08/web/predict_utils.py* 中创建了工具函数 `predict_ utils.get_current_timestamp` 来获取 `datetime` 类型的当前时间戳。`pymongo`（*https://api.mongodb.com/python/current/*）和 `bson.json_util` 都可以把 `datetime` 对象转化为 `BSON`（*http://bsonspec.org/*）表示。我们要用 `bson.json_util.dumps` 序列化 `datetime` 对象，`json.dumps` 是不行的。

然而，我们不能向 `pymongo_spark` 包传递 `Date` 对象或者 `ISODate` 对象来获取数据，因此最终还是要使用 `Timestamp` 字段的 ISO 字符串表示。ISO 8601 字符串可以直接在查询中进行大小的比较，因此在这种情况下也没有造成功能缺失：

```
def get_current_timestamp():
  iso_now = datetime.datetime.now().isoformat()
return iso_now
```

和上一节一样，我们可以使用 `curl` 测试该 API，如 *ch08/test_classification_api.sh* 所示：

```
#!/usr/bin/env bash

# 获取假设航班的延误预测
curl -XPOST 'http://localhost:5000/flights/delays/predict/classify' \
  -F 'DepDelay=5.0' \
  -F 'Carrier=AA' \
  -F 'FlightDate=2016-12-23' \
  -F 'Dest=ATL' \
  -F 'FlightNum=1519' \
  -F 'Origin=SFO' \
| json_pp
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

结果如下：

```
  % Total    %  Received  %  Xferd  Average  Speed    Time    Time    Time  Current
                                    Dload    Upload   Total   Spent   Left  Speed
100   925  100    276   100    649  13229   31107 --:--:-- --:--:-- --:--:-- 36055
{
   "FlightDate": "2016-12-23",
   "DayOfYear": 358,
   "DayOfMonth": 23,
   "Origin": "SFO",
   "FlightNum": "1519",
   "DepDelay": null,
   "Dest": "ATL",
   "Timestamp": "2016-12-12T15:30:05.272470",
   "Carrier": "AA",
   "Distance": 2139,
   "_id": {
      "$oid": "584f32fd3bf9e6056dc167ad"
   },
   "DayOfWeek": 4
}
```

最后，检查数据是否已经在 MongoDB 表中：

```
>db.prediction_tasks.find().pretty()
{
        "_id": ObjectId("584f319c3bf9e6056dc167ac"),
        "Timestamp": "2016-12-12T15:24:12.439716",
        "DepDelay": -25,
        "FlightDate": "2016-12-25",
        "FlightNum": "1519",
        "DayOfYear": 360,
        "Carrier": "DL",
        "DayOfWeek": 6,
        "Dest": "SEA",
        "Origin": "SFO",
        "DayOfMonth": 25,
        "Distance": 679
}
```

我们可以看到，API 收到的请求会在 MongoDB 表中生成相应的预测请求。现在让我们创建网页和表单，调用该 API 收集 Spark ML 批处理和实时处理的预测请求。

用于生成预测请求的前端

现在有了可以创建预测请求的 POST API，我们需要网页和表单来调用这个 API。这和前一节为 `sklearn` 回归模型创建的那一套基本相同。参见从 *ch08/web/predict_flask.py* 中节选的片段：

仅供非商业用途或交流学习使用

```
@app.route("/flights/delays/predict_batch")
def flight_delays_batch_page():
  """ 提供航班延误预测 """

  form_config = [
    {'field': 'DepDelay', 'label': 'Departure Delay'},
    {'field': 'Carrier'},
    {'field': 'FlightDate', 'label': 'Date'},
    {'field': 'Origin'},
    {'field': 'Dest', 'label': 'Destination'},
    {'field': 'FlightNum', 'label': 'Flight Number'},
  ]
  return render_template('flight_delays_predict_batch.html',
                         form_config=form_config)
```

对应的模板也和之前回归模型的模板相似。我们改动的地方用粗体标出来了：

```
{% extends "layout.html" %}
{% block body %}

  / <a href="/flights/delays/predict_batch">
    Flight Delay Prediction via Spark in Batch
      </a>

<p class="lead"style="margin: 10px; margin-left: 0px;">
  <!-- Airline name and website-->
  Predicting Flight Delays via Spark in Batch
</p>

<!-- 根据 form_config 和请求参数生成表单 -->
<form id="flight_delay_classification"
       action="/flights/delays/predict/classify"
       method="post">
  {% for item in form_config %}
    {% if 'label' in item %}
      <label for="{{item['field']}}">{{item['label']}}</label>
    {% else %}
      <label for="{{item['field']}}">{{item['field']}}</label>
    {% endif %}
      <input name="{{item['field']}}"
              style="width: 36px; margin-right: 10px;"
              value="">
      </input>
  {% endfor %}
  <button type="submit"class="btn btn-xs btn-default"style="height: 25px">
    Submit
  </button>
</form>

<div style="margin-top: 10px;">
    <p>
```


仅供非商业用途或交流学习使用

```
        Prediction Request Successful:
         <span id="result"style="display: inline-block;"></span>
        </p>
    </div>

    <script>
      // 处理表单提交
      $( "#flight_delay_classification").submit(function( event ) {

        // Stop form from submitting normally
        event.preventDefault();

          // 从页面上获取元素的值
          var $form = $( this ),
            term = $form.find( "input[name='s']").val(),
            url = $form.attr( "action");

          // 使用 post 发送数据
          var posting = $.post(
            url, $( "#flight_delay_classification").serialize()
          );

          // 把结果放在 div 里
          posting.done(function( data ) {
            $( "#result").empty().append( data );
          });
        });
      </script>
    {% endblock %}
```

我们访问 *http://localhost:5000/flights/delays/predict_batch* 测试该网页，结果如图 8-2 所示。注意：我们在页面上显示的是构建出的 JSON 格式的预测请求。在实际应用中，你的应用程序会决定要采取的正确行动。

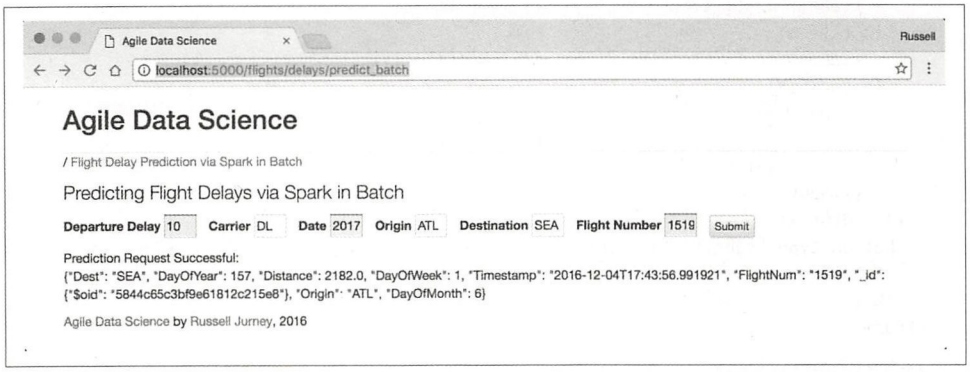


图 8-2　回归分析预测航班延误页面

仅供非商业用途或交流学习使用

构造预测请求

在继续本节内容之前，我们需要使用网页表单创建至少一个预测请求，因此请确认输入一些合理的数据，然后单击提交按钮。如果你想不出来任何机场的代码，就直接用图 8-2 里面的吧。

这样我们就搞定了批处理模式的预测请求输送工作！预测请求会从网页表单经过预测 API 一直输送到 MongoDB 中。现在我们要再一次使用 Spark，用 PySpark ML 预测代码基于 MongoDB 的 `prediction_tasks` 表中的内容进行预测。

从 MongoDB 中获取预测请求

既然我们已经在 MongoDB 中创建了预测请求，那么是时候使用 PySpark 执行真正的预测了。我们需要从 MongoDB 中获取请求，加载训练好的模型，根据请求执行预测以实现。和之前一样，为了让系统能部署到生产环境，让最终用户使用，我们需要配置所有的脚本，支持由 Airflow 从命令行直接执行。

既然模型的保存和读取都没问题了，下一步就是创建脚本去从 MongoDB 中查询一天的预测请求，并把它们保存到本地文件系统当天请求的目录里。如第 2 章所示，我们的脚本需要支持从命令行直接执行，这样 Airflow 才能控制该脚本。

从 MongoDB 中读取数据并写入文件系统的代码，如 *ch08/fetch_prediction_requests.py* 所示。脚本没有太多工作，大部分代码都是为实现使用 `spark-submit` 和 Airflow 从命令行执行而做的杂事。注意：`pymongo-spark` 让我们可以用查询语句从 MongoDB 中获取仅仅一天的记录。

让我们一段一段地检查代码。和之前一样，`main` 函数接收日期和基本路径参数。在这种情况下，日期参数可以让脚本仅对当天的预测请求实际执行预测。我们可以修改这一部分，实现每小时或者每 5 分钟之类读取一次当期预测请求（5 ~ 10 分钟差不多是批处理使用的最小的周期）：

```
#!/usr/bin/env python

import sys, os, re
import json
import datetime, iso8601

# 存入 MongoDB 中
```


仅供非商业用途或交流学习使用

```
import pymongo_spark
pymongo_spark.activate()

# 从 Airflow 中向 main() 传递日期和基本路径
def main(iso_date, base_path):

  APP_NAME = "fetch_prediction_requests.py"

  # 如果没有 SparkSession 的话，准备好环境
  try:
    sc and spark
  except NameError as e:
    import findspark
    findspark.init()
    import pyspark
    import pyspark.sql

    sc = pyspark.SparkContext()
    spark = pyspark.sql.SparkSession(sc).builder.appName(APP_NAME).getOrCreate()
```

接下来，我们使用 ISO 格式的字符串参数构造 MongoDB 查询，获取当天的数据。首先我们计算出今天和明天的日期，然后使用它们构造出 MongoDB 查询语句，放入一个 dict 对象。这个配置项对象被我们用作从 MongoDB 中读取数据的函数调用的 config 参数：

```
# 获取今天和明天的日期的 ISO 字符串表示形式来构建查询
today_dt = iso8601.parse_date(iso_date)
rounded_today = today_dt.date()
iso_today = rounded_today.isoformat()
rounded_tomorrow_dt = rounded_today + datetime.timedelta(days=1)
iso_tomorrow = rounded_tomorrow_dt.isoformat()

# 为当天数据创建 MongoDB 查询语句
mongo_query_string = """{{
  "Timestamp": {{
    "$gte": "{iso_today}",
    "$lte": "{iso_tomorrow}"
  }}
}}""".format(
  iso_today=iso_today,
  iso_tomorrow=iso_tomorrow
)
mongo_query_string = mongo_query_string.replace('\n', '')

# 使用查询语句构建 config 对象
mongo_query_config = dict()
mongo_query_config["mongo.input.query"] = mongo_query_string
```

实际调用使用 pymongo-spark 的 mongoRDD 方法来读取数据：

仅供非商业用途或交流学习使用

```
# 使用 pymongo_spark 读取当天的请求
prediction_requests = sc.mongoRDD(
  'mongodb://localhost:27017/agile_data_science.prediction_tasks',
  config=mongo_query_config
)
```

一旦读好数据，我们就可以把它转为 JSON 格式并存储。这里我们使用日期作为 *data/prediction_tasks_daily.json/* 目录中子目录名的参数。每天的数据都存在独立的目录或文件夹里，可以起到以日期为主索引的效果。这样，任何文件系统都支持对其中所存储数据的单个索引。这是我们在任何用来每天或者每小时从磁盘读取或者向磁盘保存数据的脚本里都能看到的设计模式。这也是 Airflow 调用脚本的一种模式。

注意：我们使用的是 RDD API，因此我们需要在保存数据之前调用 `os.system`，手动 `rm` 目录内容。相反，如果使用 DataFrame API，我们一般使用写覆盖模式。不论用哪种方法，我们的脚本都应该设计成把输出保存到存储区里，以覆盖前一次运行的结果。否则，系统就不能对同一个给定日期运行一次以上，这样将不利于错误的解决：

```
# 构建今天的输出路径：以日期为主键的目录结构
  today_output_path = "{}/data/prediction_tasks_daily.json/{}".format(
    base_path,
    iso_today
  )
  # 生成 JSON 记录
  prediction_requests_json = prediction_requests.map(json_util.dumps)

  # 覆盖写入今天的输出路径
  os.system("rm -rf {}".format(today_output_path))
  prediction_requests_json.saveAsTextFile(today_output_path)
if __name__ == "__main__":
  main(sys.argv[1], sys.argv[2])
```

让我们从命令行对脚本进行测试（用今天的日期替换脚本里的这个日期）：

```
python ch08/fetch_prediction_requests.py 2016-12-12 .
```

然后查看输出：

```
$ cat data/prediction_tasks.json/2016-12-12/part-00000 | json_pp
{
   "DayOfYear": 360,
   "Dest": "SFO",
   "DepDelay": -35,
   "Origin": "ATL",
   "DayOfMonth": 25,
```


仅供非商业用途或交流学习使用

```
    "FlightNum": "1519",
    "FlightDate": "2016-12-25",
    "DayOfWeek": 6,
    "Timestamp": "2016-12-12T16:27:45.463447",
    "Carrier": "AA",
    "Distance": 2139,
    "_id": {
       "$oid": "584f40813bf9e6080c27d501"
    }
  }
```

好的！我们把数据存到了日期对应的目录中。这样按日期编排的目录结构可以作为访问预测请求时的主键。这让我们可以一次处理一天的请求。现在，我们需要把一天的数据传给 Spark ML 来根据请求进行预测。

使用 Spark ML 以批处理模式进行预测

既然我们已经收集好了预测请求，是时候进行真正的预测了！参见 *ch08/make_predictions.py*。要进行预测，需要读取用 *ch08/train_spark_mllib_model.py* 保存的模型，然后把预测请求通过和训练数据相同的数据管道传递过去。

在 PySpark 中读取 Spark ML 模型

由于脚本必须是可以从命令行执行的，因此我们可以复制训练脚本的代码，用它来构建读取模型的路径。在接收命令行参数并初始化 Spark 环境之后，我们引入训练数据管道的全部模型：

```
#
# 读取管道的所有模型
#

# 读取到达延误分桶器
from pyspark.ml.feature import Bucketizer
arrival_bucketizer_path = "{}/models/arrival_bucketizer_2.0.bin".format(
  base_path
)
arrival_bucketizer = Bucketizer.load(arrival_bucketizer_path)

# 把所有字符串索引模型加载到一个 dict 对象里
from pyspark.ml.feature import StringIndexerModel
string_indexer_models = {}
for column in ["Carrier", "DayOfMonth", "DayOfWeek", "DayOfYear",
               "Origin", "Dest", "Route"]:
  string_indexer_model_path = "{}/models/string_indexer_model_{}.bin".format(
    base_path,
```


仅供非商业用途或交流学习使用

```
    column
  )
  string_indexer_model = StringIndexerModel.load(string_indexer_model_path)
  string_indexer_models[column] = string_indexer_model

# 读取数值向量组合器
from pyspark.ml.feature import VectorAssembler
vector_assembler_path = "{}/models/numeric_vector_assembler.bin".format(
  base_path
)
vector_assembler = VectorAssembler.load(vector_assembler_path)

# 读取分类器模型
from pyspark.ml.classification import RandomForestClassifier,
from pyspark.ml.classification import RandomForestClassificationModel
random_forest_model_path = \
  "{}/models/spark_random_forest_classifier.flight_delays.5.0.bin".format(
    base_path
  )
rfc = RandomForestClassificationModel.load(
  random_forest_model_path
)
```

现在我们可以读取预测请求，让它们从与训练数据相同的数据管道中流过。

使用 Spark ML 进行预测

根据日期，我们构建出当天预测请求的输入路径，把 JSON 数据读取到 DataFrame 中。为保持数据类型与训练数据一致，我们需要使用同样的表结构（除了我们要预测的 `ArrDelay` 列，以及没有使用的 `CRSDepTime` 列和 `CRSArrTime` 列）来读取数据：

```
  # 获取今天和明天的日期的 ISO 字符串表示形式来构建查询
  today_dt = iso8601.parse_date(iso_date)
  rounded_today = today_dt.date()
  iso_today = rounded_today.isoformat()

  # 构建今天的输出路径：以日期为主键的目录结构
  today_input_path = "{}/data/prediction_tasks_daily.json/{}".format(
    base_path,
    iso_today
  )

from pyspark.sql.types import StringType, IntegerType, Doubletype
from pyspark.sql.types import DateType, TimestampType
from pyspark.sql.types import StructType, StructField

  schema = StructType([
    StructField("Carrier", StringType(), True),
```

仅供非商业用途或交流学习使用

```
    StructField("DayOfMonth", IntegerType(), True),
    StructField("DayOfWeek", IntegerType(), True),
    StructField("DayOfYear", IntegerType(), True),
    StructField("DepDelay", DoubleType(), True),
    StructField("Dest", StringType(), True),
    StructField("Distance", DoubleType(), True),
    StructField("FlightDate", DateType(), True),
    StructField("FlightNum", StringType(), True),
    StructField("Origin", StringType(), True),
    StructField("Timestamp", TimestampType(), True),
])

prediction_requests = spark.read.json(today_input_path, schema=schema)
prediction_requests.show()
```

接下来，我们需要创建 Route 列：

```
#
# 添加 Route 变量取代 FlightNum
#

from pyspark.sql.functions import lit, concat
prediction_requests_with_route = prediction_requests.withColumn(
  'Route',
  concat(
    prediction_requests.Origin,
    lit('-'),
    prediction_requests.Dest
  )
)
prediction_requests_with_route.show(6)
```

现在我们把预测请求使用各个特征模型进行向量化处理。这和我们在模型训练脚本 *ch08/train_spark_mllib_model.py* 中做的几乎一模一样，仅有一处不同。我们不会删掉未处理的原始特征列，因为我们需要在存储预测的输出结果时用它们来唯一标识记录：

```
# 把字符串字段使用相应的索引模型转为索引值
for column in ["Carrier", "DayOfMonth", "DayOfWeek", "DayOfYear",
               "Origin", "Dest", "Route"]:
  string_indexer_model = string_indexer_models[column]
  prediction_requests_with_route = string_indexer_model.transform(
    prediction_requests_with_route
  )

# 向量化处理数值列：DepDelay 和 Distance
final_vectorized_features = vector_assembler.transform(
  prediction_requests_with_route
)

# 删掉称名字段的索引值
index_columns = ["Carrier_index", "DayOfMonth_index","DayOfWeek_index",
```

仅供非商业用途或交流学习使用

```
                    "DayOfYear_index", "Origin_index", "Origin_index",
                    "Dest_index", "Route_index"]
for column in index_columns:
  final_vectorized_features = final_vectorized_features.drop(column)

# 查看最终的特征
final_vectorized_features.show()
```

在准备好预测请求后，我们可以进行预测并将输出存储到当天的存储区中。删掉特征向量，这样记录会以其原始列加上预测结果列的形式返回：

```
  # 进行预测
  predictions = rfc.transform(final_vectorized_features)

  # 删除特征向量和预测的元数据，得到原始字段
  predictions = predictions.drop("Features_vec")
  final_predictions = predictions.drop("indices").drop("values") \
    .drop("rawPrediction").drop("probability")

  # 检查输出
  final_predictions.show()

  # 构建今天的输出路径：以日期为主键的目录结构
  today_output_path = "{}/data/prediction_results_daily.json/{}".format(
    base_path,
    iso_today
  )

  # 把输出保存到每天的存储区中
  final_predictions.repartition(1).write.mode("overwrite").json(
   today_output_path
  )

if __name__ == "__main__":
  main(sys.argv[1], sys.argv[2])
```

若要基于 bash 测试该脚本，只需要运行：

```
python ch08/make_predictions.py 2016-12-12 .
```

我们可以在脚本输出中看到预测（表格已为适配页面宽度而裁剪过）：

```
+-------+----------+---------+---------+--------+---------+
|Carrier|...|DepDelay|Dest|Distance|...|  Route|Prediction|
+-------+----------+---------+---------+--------+----+----+
|     DL|...|    10.0| SFO|   679.0|...|SEA-SFO|       2.0|
+-------+----------+---------+---------+--------+----+----+
```

仅供非商业用途或交流学习使用

然而，我们还是用如下命令（该脚本需要根据实际的当前日期进行修改）来检查操作实际产生的文件输出：

```
$ cat data/prediction_results_daily.json/2016-12-11/part-* | json_pp
{
  "DayOfWeek": 6,
  "Prediction": 2,
  "Carrier": "DL",
  "Origin": "SEA",
  "Distance": 679,
  "Timestamp": "2016-12-23T00:06:24.489-08:00",
  "FlightNum": "",
  "DayOfMonth": 17,
  "FlightDate": "2016-01-17",
  "DayOfYear": 17,
  "Dest": "SFO",
  "DepDelay": 10,
  "Route": "SEA-SFO"
}
```

看起来一切都很好！既然我们已经对当天的请求进行了预测，那么需要把预测结果发回 MongoDB 来让应用访问。

用 MongoDB 保存预测结果

我们的下一个任务只是整理数据，和几步之前的从 MongoDB 中获取预测请求的脚本相对应。参考 *ch08/load_prediction_results.py*，并且复习 PySpark 的 `mongo-hadoop` 文档，它是从 PySpark 连接 MongoDB 的很有帮助的参考文档。

在我们以能从命令行直接执行脚本的方式初始化脚本之后，就可以读取当天的输出数据，并存储到 MongoDB 表中：

```
# 获取今天和明天的日期的 ISO 字符串表示形式来构建查询
today_dt = iso8601.parse_date(iso_date)
rounded_today = today_dt.date()
iso_today = rounded_today.isoformat()

input_path = "{}/data/prediction_results_daily.json/{}".format(
  base_path,
  iso_today
)

# 读取 JSON 格式的文本
prediction_results_raw = sc.textFile(input_path)
prediction_results = prediction_results_raw.map(json_util.loads)

# 保存到 MongoDB 中
prediction_results.saveToMongoDB(
```

仅供非商业用途或交流学习使用

```
    "mongodb://localhost:27017/agile_data_science.prediction_results"
  )
```

我们可以在 MongoDB 控制台里检查结果：

```
> db.prediction_results.find().pretty()
{
        "_id": ObjectId("584f418d2eaf0009154e5211"),
        "FlightNum": "1519",
        "Origin": "ATL",
        "DayOfWeek": 6,
        "Dest": "SFO",
        "DepDelay": -35,
        "Prediction": 0,
        "DayOfMonth": 25,
        "Timestamp": "2016-12-12T16:27:45.463-08:00",
        "FlightDate": "2016-12-25",
        "DayOfYear": 360,
        "Carrier": "AA",
        "Distance": 2139
}
```

现在，我们的应用程序可以访问在 MongoDB 中批量预测的结果了。

在网络应用中展示批处理预测结果

现在我们的预测结果已经在 MongoDB 中可供网络应用访问，需要创建一个页面来展示它们。

参考 *ch08/web/predict_flask.py*。在开始写控制器之前，让我们先在模块的顶端引入 `datetime` 模块和 `iso8601` 模块：

```
# 日期 / 时间相关
import iso8601
import datetime
```

我们的控制器很简单。它从 GET 请求中以 slug 参数接收 ISO 格式的日期，使用该日期计算当天和明天的 ISO 日期，然后传给 MongoDB 日期范围查询来获取当天 ISO 格式日期的预测请求。最后，它会套用自身模板发回预测结果：

```
@app.route("/flights/delays/predict_batch/results/<iso_date>")
def flight_delays_batch_results_page(iso_date):
  """Serves page for batch prediction results"""

  # 获取当天和明天日期的 ISO 格式字符串来构造查询
  today_dt = iso8601.parse_date(iso_date)
  rounded_today = today_dt.date()
  iso_today = rounded_today.isoformat()
  rounded_tomorrow_dt = rounded_today + datetime.timedelta(days=1)
  iso_tomorrow = rounded_tomorrow_dt.isoformat()
```

仅供非商业用途或交流学习使用

```
    # 从 MongoDB 中获取今天的预测结果
    predictions = client.agile_data_science.prediction_results.find(
      {
        'Timestamp': {
          "$gte": iso_today,
          "$lte": iso_tomorrow,
      }
    }
  )

  return render_template(
    "flight_delays_predict_batch_results.html",
    predictions=predictions,
    iso_date=iso_date,
  )
```

我们的模板用预测结果生成了一张表格来展示结果。因为我们的预测结果是针对 0.0 ~ 2.0 这样几个分桶产生的，我们需要在模板中把分桶信息转为延误的分钟数。注意，我们可能在之前的数据流逻辑中已经这样处理过了：

```
{% extends "layout.html" %}
{% block body %}

  / <a href="/flights/delays/predict_batch/results/{{ iso_date }}">
     Flight Delay Prediction Results via Spark in Batch
  </a>

  <p class="lead"style="margin: 10px; margin-left: 0px;">

    Presenting Flight Delay Predictions via Spark in Batch
</p>

<!-- 从预测结果生成表格 -->
<table class="table">
<thead>
  <tr>
      <td>Request Timestamp</td>
      <td>Carrier</td>
      <td>Flight Date</td>
      <td>Origin</td>
      <td>Destination</td>
      <td>Distance</td>
      <td>Departure Delay</td>
      <td><span style="color: red;">Predicted Arrival Delay</span></td>
  </tr>
</thead>
<tbody>
  {% for item in predictions %}
      <tr>
```

仅供非商业用途或交流学习使用

```
                <td>{{ item['Timestamp'] }}</td>
                <td>{{ item['Carrier'] }}</td>
                <td>{{ item['FlightDate'] }}</td>
                <td>{{ item['Origin'] }}</td>
                <td>{{ item['Dest'] }}</td>
                <td>{{ item['Distance'] }}</td>
                <td>{{ item['DepDelay'] }}</td>
                <td>
                    <span style="color: red;">
                        {% if item['Prediction'] == 0.0 %}
                            On Time (0-15 Minute Delay)
                        {% elif item['Prediction'] == 1.0 %}
                            Slightly Late (15-60 Minute Delay)
                        {% elif item['Prediction'] == 2.0 %}
                            Very Late (60+ Minute Delay)
                        {% endif %}
                    </span>
                </td>
            </tr>
        {% endfor %}
    </tbody>
</table>

{% endblock %}
```

现在，通过 *http://localhost:5000/flights/delays/predict_batch/results/ 2016-12-12*（根据真实的当前日期修改链接）访问应用程序。所得页面见图 8-3。

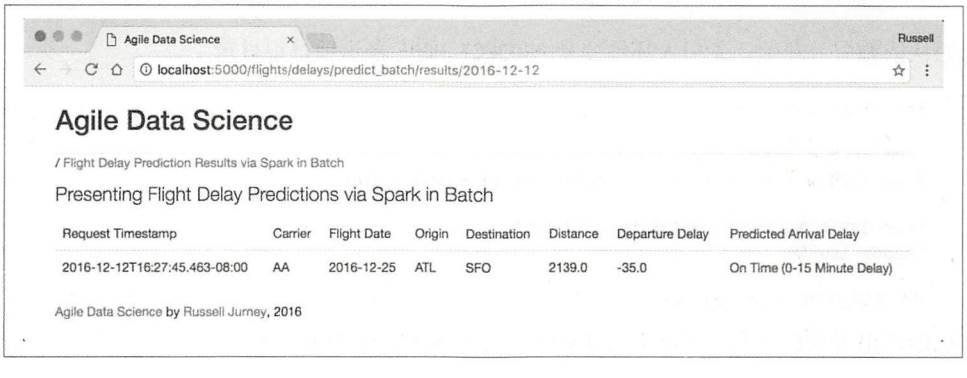


图 8-3　航班延误分类模型结果页面

实际上，这种预测可能以多种不同形式出现在你的应用中：通过发给用户的邮件，通过用户登录系统时的消息通知，作为事件或内容馈送的一部分，或者作为当天特定页面内容的一部分。也许在本书的下一版中，我们会请设计师改进这个应用程序，不过眼下这样就可以了。

仅供非商业用途或交流学习使用

用 Apache Airflow（孵化项目）自动化工作流

我们已经完成了以批处理模式部署 Spark ML 的应用和数据管道的开发部分。回顾一下，我们已经走了完整的一圈：从在应用中请求预测，回到在另一个页面中展示预测请求的结果。我们所做的所有数据处理都能从命令行执行，这样就让 Airflow 可以操作批处理模式预测系统整个数据管道的各个部分。

现在我们要用 Airflow 把我们分散的脚本整合成一个整体的可执行系统，这样我们可以通过调度任务每天执行一次，以满足生产环境的需求。注意 Airflow 是使用 Python 的 `airflow` 模块来控制的。开始先要创建一个新的 Python 脚本，使用 Airflow 把我们的脚本组合成一个完整数据流。

配置 Airflow

在本节中我们将配置 Airflow。我们在第 2 章中介绍了 Airflow，不过后来就没有用过它。如果相关记忆模糊了，可以先复习一下第 2 章中的介绍再继续（见第 59 页“使用 Apache Airflow（孵化项目）进行调度”）。

注意：我们可能会自然而然地把脚本命名为 *airflow.py*，但是这会和 Python 的 *airflow* 模块重名，导致一些问题。因此，我们创建了 airflow 目录，把脚本命名为 *setup.py*。

参考 *ch08/airflow/setup.py*，让我们逐段过一遍该脚本。

我们从依赖引入开始，先引入环境变量 `PROJECT_HOME` 表示项目根目录：

```
import sys, os, re

from airflow import DAG
from airflow.operators.bash_operator import BashOperator

from datetime import datetime, timedelta
import iso8601

PROJECT_HOME = os.environ["PROJECT_HOME"]
```

然后创建出用来创建有向无环图（DAG）和个别操作符的默认参数：

```
default_args = {
  'owner': 'airflow',
  'depends_on_past': False,
  'start_date': iso8601.parse_date("2016-12-01"),
  'email': ['russell.jurney@gmail.com'],
  'email_on_failure': True,
  'email_on_retry': True,
```

仅供非商业用途或交流学习使用

```
    'retries': 3,
    'retry_delay': timedelta(minutes=5),
}
```

创建用来创建模型的 DAG

接下来，我们实例化了一个 Airflow DAG 对象，用于特征提取和模型训练：

```
# Timedelta 为 1 表示每天运行一次
training_dag = DAG(
  'agile_data_science_batch_prediction_model_training',
  default_args=default_args
)
```

在定义操作符之前，我们先定义一个 bash_command（也就是命令行命令），该工作流中的所有 PySpark 任务都要使用它。在这条命令中，我们使用 spark-submit 来运行一个 PySpark 脚本。这条命令使用传给 BashOperator 的 params 补充参数。

我们同时使用自定义变量和 Airflow 系统变量填写 spark-submit 的参数。为脚本创建一条要使用日期的命令，再创建一条只使用基本路径的命令。自定义变量包括 Spark 主节点的主机名 {{ params.master }}，还有指向要执行的脚本的完整路径 {{ params.base_path }}/{{ params.filename }} 和脚本的基本路径参数 {{ params.base_path }}。Airflow 提供了 {{ ds }} 变量，包含 airflow 命令或者 Airflow 调度器给定的日期 / 时间值。文档中有更具体的说明，但是我们只需要了解使用 ds 变量，它可以使 PySpark 脚本和 Airflow 的 scheduler 命令或者 backfill 命令的日期功能绑定在一起：

```
# 所有 PySpark 任务都使用同样的这两条命令
pyspark_bash_command = """
spark-submit --master {{ params.master }} \
  {{ params.base_path }}/{{ params.filename }} \
  {{ params.base_path }}
"""

pyspark_date_bash_command = """
spark-submit --master {{ params.master }} \
  {{ params.base_path }}/{{ params.filename }} \
  {{ ds }} {{ params.base_path }}
"""
```

注意：在开发中，我们指定 Spark 主节点为 local，但在生产环境中，我们应该把它改为 Spark 主节点的主机名。

接下来，我们创建第一个 BashOperator，用来运行数据管道中的第一个脚本：*ch08/extract_features.py*。注意：因为我们的脚本都可以直接从命令行执行，所以通过

BashOperator 使用 spark-submit 执行它们很容易。我们把 PySpark 的 bash 命令所需的参数传过去，包括 Spark 主节点、文件名，以及基本路径等，而把时间 / 日期交给 Airflow：

```
# 为分类器模型收集训练数据
extract_features_operator = BashOperator(
task_id = "pyspark_extract_features",
  bash_command = pyspark_bash_command,
  params = {
    "master": "local[8]",
    "filename": "ch08/extract_features.py",
    "base_path": "{}/".format(PROJECT_HOME)
  },
  dag=training_dag
)
```

然后，我们为模型训练脚本 *ch08/train_spark_mllib_model.py* 创建一个 BashOperator，并指定 task_id 为 pyspark_train_classifier_model：

```
# 训练并保存分类器模型
train_classifier_model_operator = BashOperator(
  task_id = "pyspark_train_classifier_model",
  bash_command = pyspark_bash_command,
  params = {
    "master": "local[8]",
    "filename": "ch08/train_spark_mllib_model.py",
    "base_path": "{}/".format(PROJECT_HOME)
  },
  dag=training_dag
)
```

我们创建的前两个任务是紧密结合在一起的：第二个任务依赖第一个。一行代码即可确立这种关系，让两个任务一起执行：

```
# 模型训练依赖于特征提取
train_classifier_model_operator.set_upstream(extract_features_operator)
```

创建运用模型的 DAG

现在我们创建了用来创建模型的 training_dag，我们需要另一个 DAG 来每天使用模型进行预测。这个任务需要设置调度周期 schedule_interval 值为 1 的 datetime.timedelta 对象，表示任务每天执行一次：

```
daily_prediction_dag = DAG(
  'agile_data_science_batch_predictions_daily',
  default_args=default_args,
  schedule_interval=timedelta(1)
)
```


仅供非商业用途或交流学习使用

在这个 DAG 中实现任务的第一个脚本是 *ch08/fetch_prediction_requests.py*，我们把任务按含义命名为 pyspark_fetch_prediction_requests：

```
# 从 MongoDB 中获取预测请求
fetch_prediction_requests_operator = BashOperator(
  task_id = "pyspark_fetch_prediction_requests",
  bash_command = pyspark_date_bash_command,
  params = {
    "master": "local[8]",
    "filename": "ch08/fetch_prediction_requests.py",
    "base_path": "{}/".format(PROJECT_HOME)
  },
  dag=daily_prediction_dag
)
```

该段管道的第二个脚本是 *ch08/make_predictions.py*，我们把它命名为 make_predictions_operator：

```
# 运行另一个依赖于前一个脚本的简单的 PySpark 脚本
make_predictions_operator = BashOperator(
  task_id = "pyspark_make_predictions",
  bash_command = pyspark_date_bash_command,
  params = {
    "master": "local[8]",
    "filename": "ch08/make_predictions.py",
    "base_path": "{}/".format(PROJECT_HOME)
  },
  dag=daily_prediction_dag
)
```

该 DAG 中最后一个脚本为 *ch08/load_prediction_results.py*，我们把它命名为 load_prediction_results_operator：

```
# 把当天的预测结果加载到 MongoDB 中
load_prediction_results_operator = BashOperator(
  task_id = "pyspark_load_prediction_results",
  bash_command = pyspark_date_bash_command,
  params = {
    "master": "local[8]",
    "filename": "ch08/load_prediction_results.py",
    "base_path": "{}/".format(PROJECT_HOME)
  },
  dag=daily_prediction_dag
)
```

现在我们已经为 daily_prediction_dag 创建了操作，我们需要用正式的依赖关系把它们结合到一起。三个脚本间有两个依赖关系，这次我们自上而下设置依赖关系，而不是像之前那样自下而上。两种方式的最终结果是一样的，不论自上而下还是自下而上：

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

```
# 设置自上而下依赖
fetch_prediction_requests_operator.set_downstream(make_predictions_operator)
make_predictions_operator.set_downstream(load_prediction_results_operator)
```

这样我们便讲完了 Airflow 配置脚本。下面我们要运行脚本并了解 `airflow` 命令。

使用 Airflow 管理与执行 DAG 和任务

首先，我们需要使用 Airflow 系统配置脚本，验证一切都能正确解析，然后再测试各个任务，然后再整体测试各个 DAG。

把 Airflow 脚本链接放到 Airflow DAG 目录下。为了运行脚本并往 Airflow 里添加 DAG，我们需要把脚本链接放到 Airflow 的 *dags/* 目录下，也就是 *~/airflow/dags/*：

```
ln -s $PROJECT_HOME/ch08/airflow/setup.py ~/airflow/dags/setup.py
```

在执行之前验证链接生效。注意实际的路径可能会因环境变量 `$PROJECT_HOME` 的值而有所不同：

```
$ ls -lah ~/airflow/dags/

total 16
drwxr-xr-x 5 rjurney staff 170B Dec 12 18:07 .
drwxr-xr-x 13 rjurney staff 442B Dec 12 18:07 ..
drwxr-xr-x 4 rjurney staff 136B Dec 12 18:07 __pycache__
lrwxr-xr-x 1 rjurney staff 62B Dec 3 23:02 airflow_test.py -> \
  /Users/rjurney/Software/Agile_Data_Code_2/ch02/airflow_test.py
lrwxr-xr-x 1 rjurney staff 63B Dec 12 18:07 setup.py -> \
  /Users/rjurney/Software/Agile_Data_Code_2/ch08/airflow/setup.py
```

执行 Airflow 配置脚本。现在可以在 *~/airflow/dags/* 中适时执行脚本，把它添加到 Airflow 系统了。输出简单，没有太多信息，不过没有报错便表明一切正常：

```
$ python ~/airflow/dags/setup.py

[2016-12-12 18:10:53,413] {__init__.py:36} INFO - Using executor
              SequentialExecutor
```

从命令行查询 Airflow。我们可以使用 `airflow list_dags` 命令来查看 Airflow 中已配置的 DAG。这里显示了我们在第 2 章中配置的测试 DAG，以及刚才在脚本中定义的两个 DAG：

```
$ airflow list_dags

agile_data_science_airflow_test
agile_data_science_batch_prediction_model_training
agile_data_science_batch_predictions_daily
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

现在我们可以对每个 DAG 使用 list_tasks 命令来查看组成该 DAG 的任务。首先让我们检查 agile_data_science_batch_prediction_model_training：

```
$ airflow list_tasks agile_data_science_batch_prediction_model_training

pyspark_extract_features
pyspark_train_classifier_model
```

接下来，让我们对 agile_data_science_batch_predictions_daily 使用 list_tasks：

```
$ airflow list_tasks agile_data_science_batch_predictions_daily

pyspark_fetch_prediction_requests
pyspark_load_prediction_results
pyspark_make_predictions
```

在 Airflow 中测试任务。现在我们可以使用 airflow 命令测试我们刚才所创建的各个任务的执行过程。参阅 *ch08/test_airflow.sh*。使用阅读这里的日期替换这里当天的日期，或者通过 export ISO_DATE=`date "+%Y- %m-%d"` 设置日期环境变量，并把该环境变量 $ISO_DATE 放到每条命令中替换日期，和我们在 *ch08/test_airflow.sh* 中所做的一样：

```
airflow test agile_data_science_batch_prediction_model_training \
  pyspark_extract_features 2016-12-12
```

你会看到 spark-submit 的大量输出，但在那之前有一些东西是有用的。Airflow 会显示任务尝试次数（这里是 4 次中的第 1 次）以及实际执行的完整 spark-submit 命令。你可以使用这条命令来调试遇到的任何问题。如果执行成功，系统会标明命令执行的返回为 0：

```
--------------------------------------------------------------------------------
Starting attempt 1 of 4
--------------------------------------------------------------------------------
[2016-12-12 19:36:13,491] {models.py:1219} INFO - Executing <Task(BashOperator):
  pyspark_extract_features> on 2016-12-12 00:00:00
[2016-12-12 19:36:13,502] {bash_operator.py:55} INFO - tmp dir root location:
/var/folders/0b/74l_65015_5fcbmbdz1w2xl40000gn/T
[2016-12-12 19:36:13,503] {bash_operator.py:64} INFO - Temporary script location:
/var/folders/0b/74l_65015_5fcbmbdz1w2xl40000gn/T/airflowtmpymttr3lj//var/ \
  folders/0b/74l_65015_5fcbmbdz1w2xl40000gn/T/airflowtmpymttr3lj/ \
    pyspark_extract_featurestk4dvpse
[2016-12-12 19:36:13,503] {bash_operator.py:65} INFO - Running command:
spark-submit --master local[8] /Users/rjurney/Software/Agile_Data_Code_2//ch08/
extract_features.py 2016-12-12
/Users/rjurney/Software/Agile_Data_Code_2/

...
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

```
[2016-12-12 20:06:22,971] {bash_operator.py:80} INFO - Command exited with return
  code 0
```

现在为同一个 DAG agile_data_science_batch_prediction_model_training 中的任务 pyspark_train_classifier_model 进行同样的尝试。然后为 DAGagile_data_science_batch_predictions_daily 中的每个任务重复这一步骤，从如下命令开始：

```
airflow test agile_data_science_batch_predictions_daily \
pyspark_fetch_prediction_requests
```

在继续前进之前，执行 *ch08/test_airflow.sh* 中剩余的测试命令。这证明我们所有的任务都是正确的！但是这些任务能共同工作吗？

在 Airflow 中测试 DAG。现在我们已经测试好了任务，还需要整体测试 DAG。我们可以配置单天使用 airflow backfill 命令来实现。单独执行 airflow backfill 来查看命令选项。注意：backfill 也可以做它名字所表示的任务，即“回填”数据中的洞，比如，当创建新数据管道需要填入历史数据时：

```
[2016-12-12 19:46:22,679] {__init__.py:36} INFO - Using executor
            SequentialExecutor
usage: airflow backfill [-h] [-t TASK_REGEX] [-s START_DATE] [-e END_DATE]
                        [-m] [-l] [-x] [-a] [-i] [-I] [-sd SUBDIR]
                        [--pool POOL] [-dr]
                        dag_id
airflow backfill: error: the following arguments are required: dag_id
```

测试 DAG agile_data_science_batch_prediction_model_training 的命令如下所示。注意，这可能要执行几分钟的时间：

```
airflow backfill -s 2016-12-12 -e 2016-12-12
    agile_data_science_batch_prediction_model_training
```

测试 DAG agile_data_science_batch_predictions_daily 的命令：

```
airflow backfill -s 2016-12-12 -e 2016-12-12
           agile_data_science_batch_predictions_daily
```

backfill 限制了调试级别的输出，但它还是为每次执行打印了日志文件的路径。你可以用 cat 或者 tail -f 查看日志文件获取更多信息。假如两个任务运行都没遇到问题，我们就可以检查 Airflow 网页界面来查看一些情况了。

在 Airflow 网页界面上监视任务。如果还没有启动 Airflow 调度器和 Airflow 网页界面，先通过如下命令启动：

```
airflow scheduler -D
airflow webserver -D
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

现在访问位于 *http://localhost:8080/admin/#/* 的 Airflow 界面（见图 8-4）。如果你看到的是 Zeppelin 界面，则先关闭 Zeppelin 守护进程：

```
zeppelin/bin/zeppelin-daemon.sh stop
```

如果 DAG 被标记为不活跃状态，则单击旁边的刷新按钮，它们就会刷新为活跃状态。

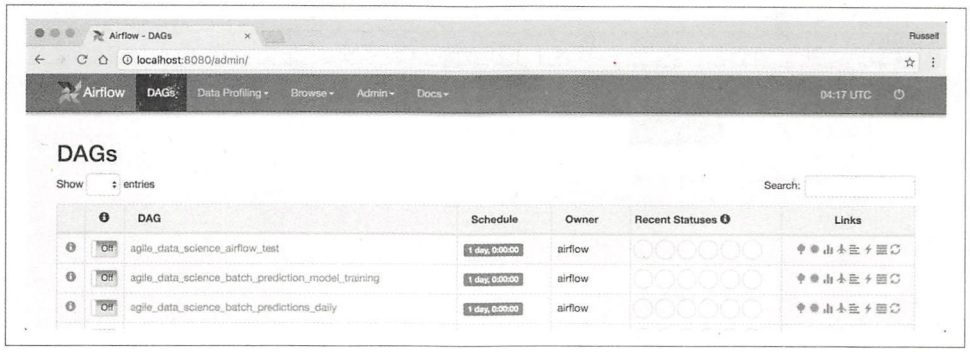


图 8-4　Airflow 管理员主页的 DAG 列表

单击名为 `agile_data_science_batch_predictions_daily` 的 DAG，可以看到该 DAG 的图形显示（见图 8-5）。注意：这张图展示了我们在 *ch08/airflow/setup.py* 中为 DAG `agile_data_science_batch_predictions_daily` 所配置的依赖关系。

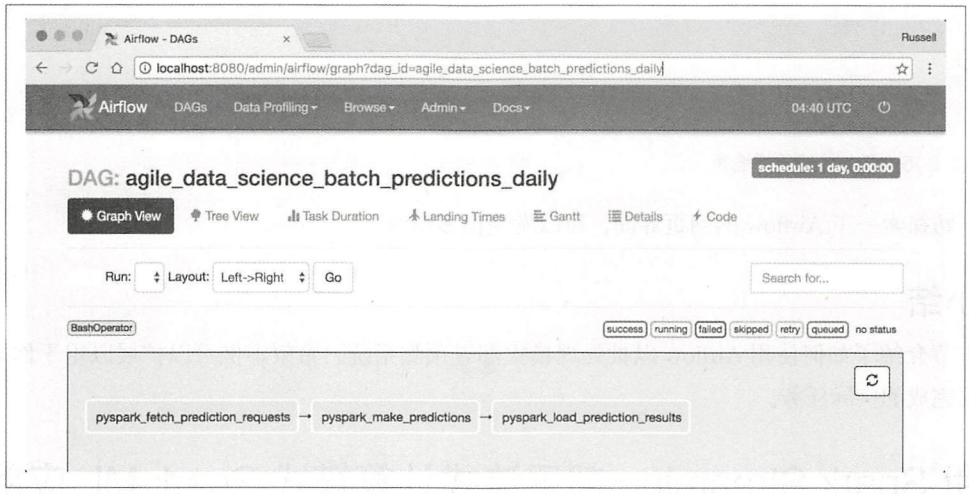


图 8-5　Airflow DAG 页面

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

单击任务持续时间（Task Duration）链接（*http://localhost:8080/admin/airflow/duration?root=&days=30&dag_id=agile_data_science_batch_predictions_daily*），打开的页面会显示一个展示回填操作运行时间的图表（见图 8-6）。

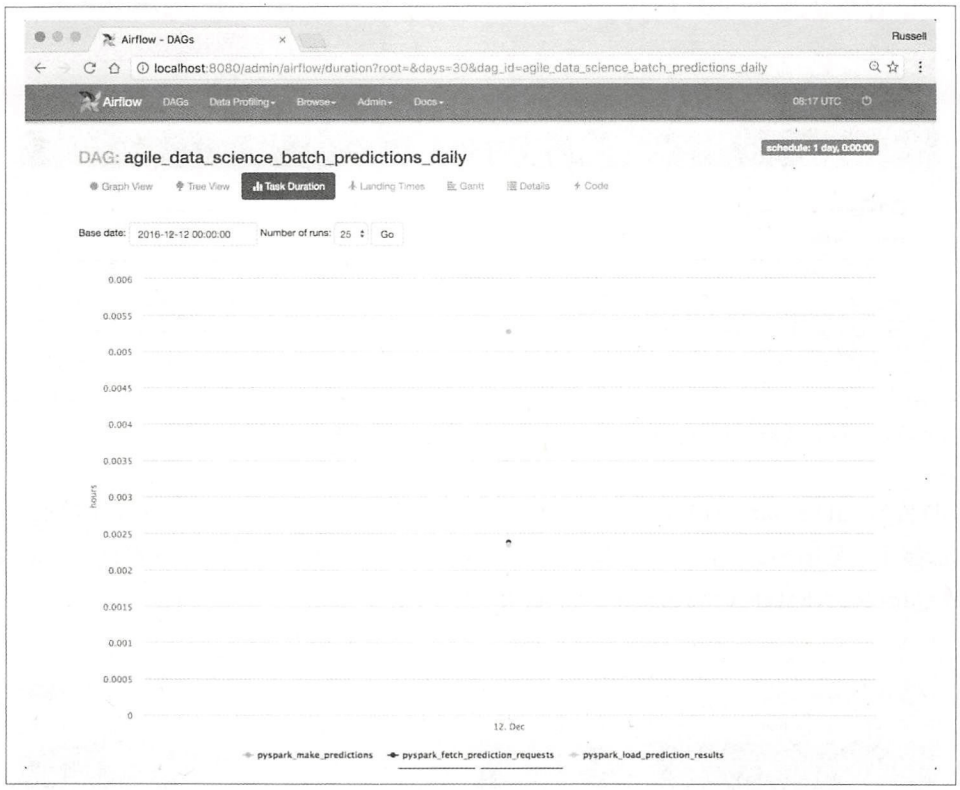


图 8-6　任务持续时间图表

不妨探索一下 Airflow 的网页界面，可以发现很多东西。

小结

本节介绍了如何使用 Airflow 以批处理模式部署预测系统。希望本例可以扩展以用于你需要完成的实际任务。

用 Spark Streaming 部署流式计算模式 Spark ML 应用

随着 Apache Kafka 的兴起，Spark Streaming 在“准实时”处理数据方面越来越流行。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

Spark Streaming 的工作流会复用前一节中实现的训练、存储、加载分类器模型的代码。但是其他内容还是会不大一样的，首先就是我们创建预测请求的方式。

在这种情况下，网络应用会在需要进行预测时发出一个 Kafka 事件，通过 Kafka 集群，Spark Streaming 进程会收到该事件，并使用从硬盘上读取的模型进行特征向量化和预测。然后把预测结果写入数据库，原始的网络应用程序将在那里读取并发表结果。

请注意，这种部署方式的一个重要限制就是只能同时使用一个模型。这可能会成为一些应用的瓶颈，比如基于内容进行预测的预测系统，每个用户都需要有独立的模型，而在 Spark Streaming 进程中无法循环调用多个模型。

在生产环境中收集训练数据

我们要复用前一节以批处理模式部署预测系统的收集训练数据的代码和 Airflow 配置。你可以参考 *ch08/extract_features.py* 并复习第 235 页“在生产环境中收集训练数据”的内容。即使以 Spark Streaming 部署预测系统，在生产环境里收集训练数据还是使用批处理模式的 PySpark 和 Airflow。

Spark ML 模型的训练、存储、读取

同样，我们还要复用前一节以批处理模式部署预测系统的训练和保存模型的代码以及 Airflow 配置。你可以参考 *ch08/train_spark_mllib_model.py* 和第 237 页“Spark ML 模型的训练、存储与加载”的内容。即使以 Spark Streaming 部署预测系统，我们依然使用 Airflow 来自动化模型的训练与保存，然后部署到 Spark Streaming 中。图 8-7 展示了我们的后端架构。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

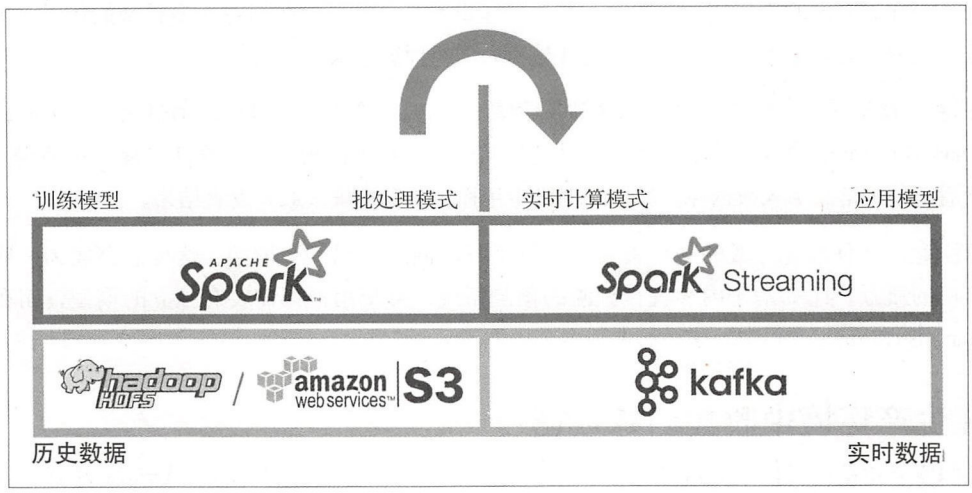


图 8-7　后端架构

发送预测请求到 Kafka

为了把预测任务和相关数据传给 Spark ML 脚本，我们需要生成表示要进行预测的 Kafka 事件。网络应用可以利用表单和对应的表单控制器生成 Kafka 消息形式的预测请求来简单实现。

这与通过网络 API 部署的 `scikit-learn` 相反，在那种情况下预测结果可以直接在网络应用内计算得到。在真实世界中，这可能适用于各种各样的情况，因为任何超过几分之一秒的计算都最好通过 Kafka 部署。Kafka 的工作节点正是处理那些会导致延迟的负载高峰的地方——Kafka 设计的初衷就是处理这种变化的负载，而网络应用不是。

在这种情况下，网络应用会在 Kafka 的一个主题下发出一条请求。然后，Spark Streaming 的一个微型批处理会获取预测请求，传递给 PySpark ML，后者会生成预测结果并保存到 MongoDB 表中。然后预测请求页面会展示预测请求对应的预测结果。

配置 Kafka

我们要和第 2 章一样先配置 Kafka 然后再继续。你可能需要参考第 54 页“使用 Apache Kafka 分发流数据”重拾 Kafka。如果你已经做过这些步骤，并且还没有重启电脑，那么你可以跳过已经完成的那些步骤。

仅供非商业用途或交流学习使用

我们需要启动 Zookeeper 和 Kafka 服务器端程序，然后为预测请求创建一个 Kafka 主题。

启动 Zookeeper。Zookeeper 帮助我们管理 Kafka，所以我们需要先启动它。为 Zookeeper 新开一个控制台，然后运行：

```
kafka/bin/zookeeper-server-start.sh kafka/config/zookeeper.properties
```

启动 Kafka 服务器端。现在，在一个新的控制台里，运行 Kafka 服务器端程序：

```
kafka/bin/kafka-server-start.sh kafka/config/server.properties
```

创建主题。打开一个新的控制台。我们将使用它来运行各种 Kafka 命令，然后让它运行预测请求主题的控制台消费者程序。

Kafka 消息是按主题分组的，所以我们在通过 Kafka 发送消息之前，需要先创建一个主题：

```
kafka/bin/kafka-topics.sh \
    --create \
    --zookeeper localhost:2181 \
    --replication-factor 1 \
    --partitions 1 \
    --topic flight_delay_classification_request
```

我们会看到如下的消息：

```
Created topic "flight_delay_classification_request".
```

验证预测请求主题创建成功。我们可以用主题列表命令看到我们创建的主题：

```
$ kafka/bin/kafka-topics.sh --list --zookeeper localhost:2181
flight_delay_classification_request
test
```

我们需要监视这个主题，因此让我们运行 `flight_delay_classification_request` 主题的控制台消费者程序：

```
kafka/bin/kafka-console-consumer.sh \
    --bootstrap-server localhost:9092 \
    --topic flight_delay_classification_request \
    --from-beginning
```

这样就好了。现在 Kafka 已经可以发送和接收预测请求了，等到请求来的时候我们就能看到！现在我们配置网络应用以发出请求。

从 Flask API 向 Kafka 传递推荐任务

在 *ch08/web/flask_predict.py*（*https://github.com/dpkp/kafka-python*）中，我们要利用 `kafka-python` 生成生产者，从网络应用发出 Kafka 事件。

首先我们需要引入 `kafka-python`，配置 `KafkaProducer` 对象，用它发出事件。我们把这部分代码写在脚本开头，以便别的控制器使用 Kafka。我们还要引入 Python 包 `uuid`（*https://docs.python.org/3.1/library/uuid.html*），它可以为我们的预测请求创建一个唯一的 ID：

```
# 配置 Kafka
from kafka import KafkaProducer, TopicPartition
producer = KafkaProducer(bootstrap_servers=['localhost:9092'],api_version=(0,10))
PREDICTION_TOPIC = 'flight_delay_classification_request'

import uuid
```

接下来，我们基于批处理模式的预测请求 API 稍做修改，为现在的预测请求创建一个新的 API。我们会向 Kafka 主题发出 JSON 格式的请求，而不是把请求插到 MongoDB 中。这些请求都有各自的全局唯一标识符（UUID）可供区分——UUID 是一个长到难以发生重复的随机字符串：

```
# 让 API 只接收 POST 请求，这样搜索引擎不会访问到
@app.route("/flights/delays/predict/classify_realtime", methods=['POST'])
def classify_flight_delays_realtime():

  # 定义要处理的表单字段
  """ 分类分析航班延误的 POST API"""
  api_field_type_map = \
    {
      "DepDelay": float,
      "Carrier": str,
      "FlightDate": str,
      "Dest": str,
      "FlightNum": str,
      "Origin": str
    }

  # 从表单对象中获取各字段的值
  api_form_values = {}
  for api_field_name, api_field_type in api_field_type_map.items():
    api_form_values[api_field_name] = request.form.get(
      api_field_name, type=api_field_type
    )

  # 设置无须修改的值，不包括日期
  prediction_features = {}
```

```
    for key, value in api_form_values.items():
      prediction_features[key] = value

    # 设置计算得出的值
    prediction_features['Distance'] = predict_utils.get_flight_distance(
      client, api_form_values['Origin'],
      api_form_values['Dest']
    )

    # 把日期转为 DayOfYear、DayOfMonth、DayOfWeek
    date_features_dict = predict_utils.get_regression_date_args(
      api_form_values['FlightDate']
    )
    for api_field_name, api_field_value in date_features_dict.items():
      prediction_features[api_field_name] = api_field_value

    # 添加时间戳
    prediction_features['Timestamp'] = predict_utils.get_current_timestamp()

    # 为这条消息创建唯一的 ID
    unique_id = str(uuid.uuid4())
    prediction_features['UUID'] = unique_id

    message_bytes = json.dumps(prediction_features).encode()
    producer.send(PREDICTION_TOPIC, message_bytes)

    response = {"status": "OK", "id": unique_id}
    return json_util.dumps(response)
```

我们可以使用 curl 测试这个 API，然后用上一节中配置的控制台消费者程序进行监控：

```
curl -XPOST 'http://localhost:5000/flights/delays/predict/classify_realtime' \
    -F 'DepDelay=5.0' \
    -F 'Carrier=AA' \
    -F 'FlightDate=2016-12-23' \
    -F 'Dest=ATL' \
    -F 'FlightNum=1519' \
    -F 'Origin=SFO' | json_pp
```

返回数据包含状态码和 UUID：

```
% Total    %  Received  %  Xferd  Average  Speed    Time    Time    Time  Current
                                   Dload   Upload   Total   Spent   Left  Speed
100   711  10062  100   649   7322  76650   --:--:--  --:--:--  --:--:--   126k
{
    "status": "OK",
    "id": "fbb5b61c-2c7b-4db6-a22f-dae270c59797"
}
```

这条请求应该和 UUID 一起出现在控制台消费者程序中：

```
{
 "Distance": 2139.0,
 "Carrier": "AA",
 "DayOfYear": 358,
 "UUID": "fbb5b61c-2c7b-4db6-a22f-dae270c59797",
 "DayOfMonth": 23,
 "Origin": "SFO",
 "FlightNum": "1519",
 "Dest": "ATL",
 "DepDelay": 5.0,
 "DayOfWeek": 4,
 "FlightDate": "2016-12-23",
 "Timestamp": "2016-12-13T20:21:29.233822"
}
```

就是这样！我们成功地在 Kafka 中生成了预测请求事件。

生成预测请求的前端

现在我们需要一个前端来创建预测请求并展示预测结果。这和本章前半部分为 `scikit-learn` 回归分析网络服务实现的前端看起来相似，但是多了一个新变化：轮询表单。

轮询请求和 LinkedIn 关系网 InMap。这个功能区别于我们之前实现的使用简单的 AJAX（*https://mzl.la/2oYXz57*）请求 POST 表单并显示返回内容的功能。这种表单介于本章中创建的两个表单之间（第 232 页“在产品中使用 API”中创建的在网络应用中直接调用 `scikit-learn` 模型并立即把输出显示在页面上的实时表单，以及第 242 页“用于生成预测请求的前端”中创建的提交预测请求但是没有结果返回的表单，因为输出会在批量处理完成后显示在另一个页面上）。

这一次我们的表单所提交的请求会有返回，但不是立刻就有。首先它会收到一条返回表示预测请求已经收到了。这会让客户端发送另一个请求到另外一个接口去请求预测结果。如果预测还没有完成，返回结果会告诉客户端再等待一段时间重发请求。同时，客户端会在页面上显示“处理中”的消息。当结果最终产生，才会显示在页面上。

图 8-8 展示了我们这个应用的前端架构。

仅供非商业用途或交流学习使用

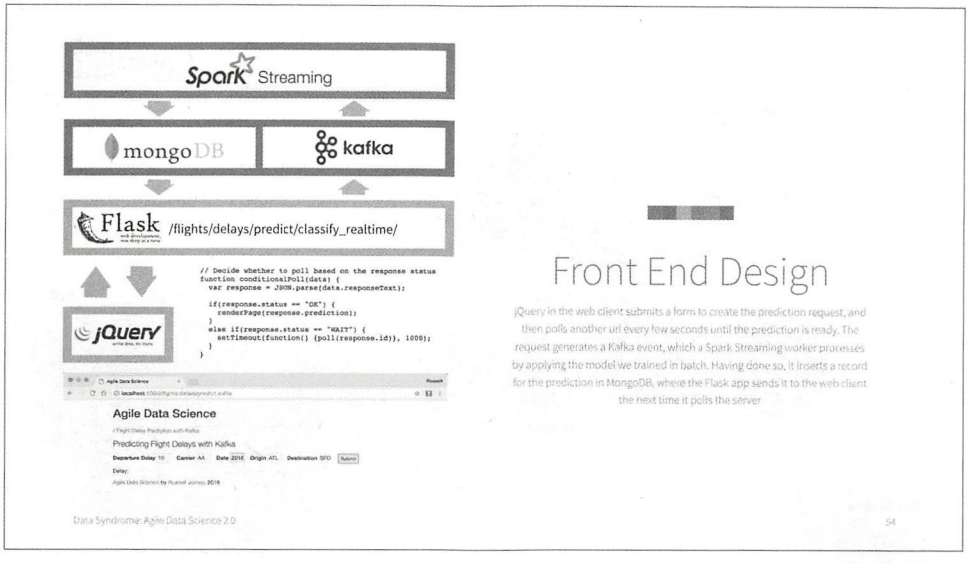


图 8–8　前端架构

许多真实的产品使用了这种模式进行数据处理和预测。例如，LinkedIn 关系网 InMap（*http://oreil.ly/2o7Gwyi*）就是用了这个模式。在用户通过 LinkedIn API 授权后，客户端会发出异步请求，要求生成用户关系网的可视化图像。这会生成一个准备关系网图像的请求。“渲染场”中的服务器会为每个关系网生成一个强制定向的分布图，并为每个用户渲染背景图像。当关系网图生成出来时，渲染节点会在数据库中创建一条记录。同时，客户端会不断轮询第二个接口，并在每次请求之间有所间隔，以等待关系网图。屏幕上的消息显示关系网图正在绘制中，直到关系网图最终传输到客户端并显示出来为止（见图 8-9）。这种社交功能鼓励用户分享关系网图，形成分发产品的病毒式传播。

仅供非商业用途或交流学习使用

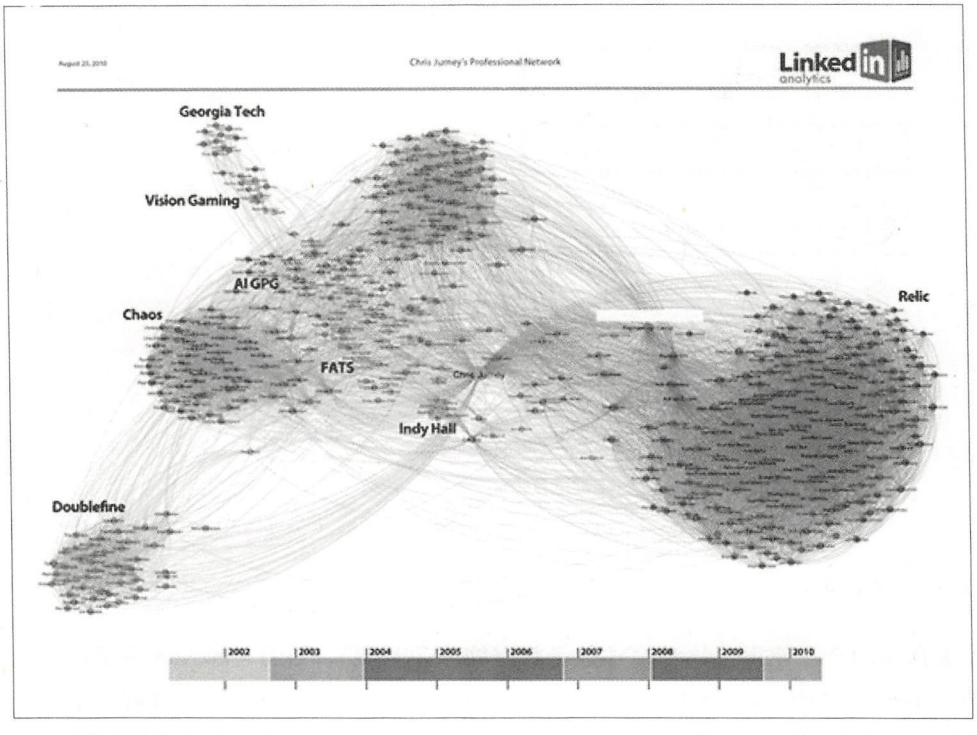


图 8-9　LinkedIn 关系网 InMap 原型

我们将使用轮询表单来显示航班延误预测结果。这需要两个接口及其对应的控制器，以及一些简单的 JavaScript 脚本。

页面控制器。如 *ch08/web/predict_flask.py* 所示，我们定义了一个简单的控制器为预测页面提供模板：

```
@app.route("/flights/delays/predict_kafka")
def flight_delays_page_kafka():
  """提供带有轮询表单的航班延误预测页面"""

  form_config = [
    {'field': 'DepDelay', 'label': 'Departure Delay'},
    {'field': 'Carrier'},
    {'field': 'FlightDate', 'label': 'Date'},
    {'field': 'Origin'},
    {'field': 'Dest', 'label': 'Destination'},
  ]

  return render_template(
   'flight_delays_predict_kafka.html', form_config=form_config
  )
```

仅供非商业用途或交流学习使用

提供预测结果的 API 控制器。我们还需要一个简单的控制器在预测完成时提供结果，根据 MongoDB 中是否有给定 UUID 对应的记录进行判断。这就是简单的 CRUD 操作，这也是一般网络应用中最常见的场景：

```
@app.route("/flights/delays/predict/classify_realtime/response/<unique_id>")
def classify_flight_delays_realtime_response(unique_id):
  """ 为轮询请求提供预测结果 """

  prediction = \
    client.agile_data_science.flight_delay_classification_response.find_one(
      {
        "id": unique_id
      }
    )

  response = {"status": "WAIT", "id": unique_id}
  if prediction:
    response["status"] = "OK"
    response["prediction"] = prediction

return json_util.dumps(response)
```

我们可以用 curl 验证它按预期方式工作：

```
curl \
  'http://localhost:5000/flights/delays/predict/classify_realtime \
    /response/EXAMPLE_UUID_g3t03qtq3t' | json_pp
```

结果如下：

```
  % Total    %  Received  %  Xferd  Average  Speed    Time    Time    Time  Current
                                     Dload   Upload    Total   Spent   Left  Speed
100     51   100     51      0     0   7834       0 --:--:-- --:--:-- --:--:-- 25500

{
  "id": "EXAMPLE_UUID_g3t03qtq3t",
  "status": "WAIT"
}
```

现在往 MongoDB 中插入一条对应该 UUID 的数据：

```
db.flight_delay_classification_response.insert(
  {
    id: "EXAMPLE_UUID_g3t03qtq3t",
    prediction: {"test": "data"}
  }
)
```

然后再试一次：

```
curl 'http://localhost:5000/flights/delays/predict/classify_realtime/ \
   response/EXAMPLE_UUID_g3t03qtq3t' | json_pp
```

我们的记录直接变成了返回中的预测结果部分：

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

```
% Total    %  Received  %  Xferd  Average  Speed    Time    Time    Time  Current
                                   Dload   Upload    Total   Spent   Left  Speed
100   175  100    175      0     0  31605       0 --:--:-- --:--:-- --:--:-- 87500
{
  "id" : "EXAMPLE_UUID_g3t03qtq3t",
  "status" : "OK",
  "prediction" : {
     "_id" : {
        "$oid" : "5850dc50ebc402b548a0234c"
     },
     "id" : "EXAMPLE_UUID_g3t03qtq3t",
     "prediction" : {
        "test" : "data"
     }
  }
}
```

创建轮询表单的模板。这个控制器的模板是 *ch08/web/templates/flight_delays_predict_kafka.html*，它是我们从 *ch08/web/ templates/flight_delays_predict.html* 复制过来并进行适当修改得到的。打开它并跟上：

```
{% extends "layout.html" %}
{% block body %}
  <!-- 导航提示 -->
  / <a href="/flights/delays/predict_kafka">
      Flight Delay Prediction with Kafka
    </a>

  <p class="lead"style="margin: 10px; margin-left: 0px;">

    Predicting Flight Delays with Kafka
  </p>

  <!-- 根据 form_config 和请求参数生成表单 -->
  <form id="flight_delay_classification"
         action="/flights/delays/predict/classify_realtime"
         method="post">
    {% for item in form_config %}
      {% if 'label' in item %}
        <label for="{{item['field']}}">{{item['label']}}</label>
      {% else %}
        <label for="{{item['field']}}">{{item['field']}}</label>
      {% endif %}
        <input name="{{item['field']}}"
                style="width: 36px; margin-right: 10px;"
                value="">
        </input>
  {% endfor %}
  <button type="submit"class="btn btn-xs btn-default"style="height: 25px">
    Submit
  </button>
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

```
    </form>

    <div style="margin-top: 10px;">
        <p>Delay: <span id="result"style="display: inline-block;"></span></p>
    </div>

    <script src="/static/js/flight_delay_predict_polling.js"></script>
  {% endblock %}
```

模板本身很简单，真正的工作在 JavaScript 文件 *ch08/web/static/flight_delay_predict_polling.js* 中。让我们一段一段地看。我们把任务分割到一系列函数里，保持每个函数干净简单。

和我们之前所做的一样，使用 `jQuery.submit`（*https://api.jquery.com/submit/*）在我们提交 HTML 表单时附加一个函数。利用这个函数，我们处理表单的输入并发给表单的接口 */flights/delays/predict/classify_realtime*。一旦收到表示有效预测请求被提交的返回，我们就开始使用返回消息中给预测请求分配的唯一 ID 来轮询另一个接口：

```
// 为表单附加一个提交处理函数
$( "#flight_delay_classification").submit(function( event ) {

  // 阻止正常的表单提交
  event.preventDefault();

  // 从页面上获取一些元素的值
  var $form = $( this ),
    term = $form.find( "input[name='s']").val(),
    url = $form.attr( "action");

  // 使用 post 发送数据
  var posting = $.post(
    url,
    $( "#flight_delay_classification").serialize()
  );

  // 提交表单，解析返回内容
  posting.done(function( data ) {
    response = JSON.parse(data);

  // 如果返回状态为 OK，则输出消息让用户等待并开始轮询
  if(response.status == "OK") {
    $( "#result").empty().append( "Processing...");

    // 每隔 1 秒钟，轮询返回 URL，直至收到返回为止
    poll(response.id);
    }
  });
});
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

轮询是由 poll 函数处理的。poll 函数接收预测请求的 ID，根据 ID 生成 */flights/delays/predict/classify_realtime/response/* 返回接口对应的 URL，只要把 ID 作为 slug 放在接口的末尾作为参数即可。它向该 URL 提交第一个异步 GET 请求，把返回结果导向函数 conditionalPoll：

```
// 轮询预测 URL
function poll(id) {
  var responseUrlBase = "/flights/delays/predict/classify_realtime/response/";
  console.log("Polling for request id "+ id + "...");

  // 把 UUID 接在 URL 后面作为 slug 参数
  var predictionUrl = responseUrlBase + id;

  $.ajax(
  {
    url: predictionUrl,
    type: "GET",
    complete: conditionalPoll
  });
}
```

conditionalPoll 顾名思义，就是在返回为 WAIT 时再次轮询接口，而在返回状态为 OK 时通过函数 renderPage 在页面上渲染返回结果：

```
// 根据返回状态判断是否继续轮询
function conditionalPoll(data) {
  var response = JSON.parse(data.responseText);

  if(response.status == "OK") {
    renderPage(data);
  }
  else if(response.status == "WAIT") {
    setTimeout(function() {poll(response.id)}, 1000);
  }
}
```

renderPage 则非常简单。它在页面上输出预测返回值，位置与实时的 scikit-learn 预测前端一样：

```
// 根据分类界限 [-float("inf"), -15.0, 0, 30.0, float("inf")]
// 在页面上渲染返回结果
function renderPage(response) {

  var displayMessage;

  if(response.Prediction == 0) {
    displayMessage = "Early (15+ Minutes Early)";
  }
  else if(response.Prediction == 1) {
    displayMessage = "Slightly Early (0-15 Minute Early)";
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

```
    }
    else if(response.Prediction == 2) {
      displayMessage = "Slightly Late (0-30 Minute Delay)";
    }
    else if(response.Prediction == 3) {
      displayMessage = "Very Late (30+ Minutes Late)";
    }
    $( "#result").empty().append( displayMessage );
  }
```

这样，我们就完成了基于 Kafka 预测的前端工作！现在让我们做一些测试。

提交预测请求

为了测试页面，请访问 *http://localhost:5000/flights/delays/predict_kafka* 并打开 JavaScript 控制台。现在，输入一些测试数据并且提交表单。确保所有的字段都填好。你应该会看到一个提示等待的消息，以及向预测结果 URL 发出的每秒一次的请求，见图 8-10。

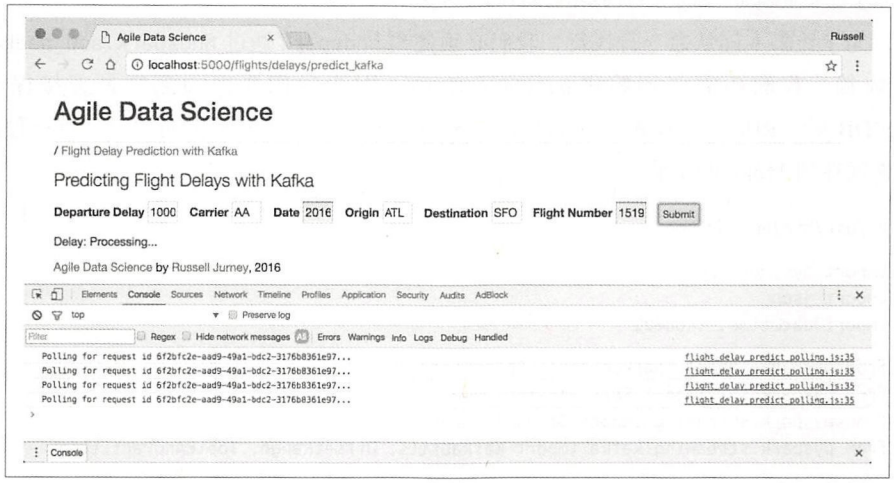


图 8-10　基于 Kafka 的航班延误预测页面

现在我们的模型已经准备妥当，预测请求都能生成对应的 Kafka 事件，前端也准备好显示结果了，我们可以开始使用 Spark Streaming 进行预测了！

用 Spark Streaming 进行预测

现在我们已经创建了通过 Kafka 提交预测请求的前端，也实现了在网页上显示预测结果，我们需要完成中间的部分，也就是用 PySpark Streaming 处理 Kafka 事件，把结果插入 MongoDB 中以供前端渲染。注意：你可以在 Spark Streaming 编程指导里学到更多有关部

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

署 PySpark Streaming 的内容。

参考 *ch08/make_predictions_streaming.py*（*http://bit.ly/2o7trFf*）。我们用类似于使用批处理模式的方式初始化系统，但是除了 `SparkSession`（*http://bit.ly/2oOkAqO*），还要初始化 `StreamingContext`（*http://bit.ly/2nRGlqX*）。只要它们来自同一个 `SparkContext`（*http://bit.ly/2oTAQoV*），就能共同工作（尽管我们会发现 Spark Streaming 基本上是基于 RDD 工作的）。

注意：我们的 `main` 函数这次只接收一个参数 `base_path`。不再需要日期了，因为脚本会处理它看到的所有 Kafka 事件。

为了执行 Spark Streaming 脚本，我们需要引入 Spark Streaming 包，现在的名字是 `org.apache.spark:spark-streaming-kafka-0-8_2.11:2.1.0`（在你读到这里的时候，版本可能已经有了更新）。我们可以在开发时使用 PySpark 控制台获取它：

```
PYSPARK_DRIVER_PYTHON=ipython pyspark --packages \
org.apache.spark:spark-streaming-kafka-0-8_2.11:2.1.0
```

然而，为了让脚本能从命令行执行，我们必须使用 `findspark` 通过 `findspark.add_packages` 引入依赖。我们还需要初始化 `pymongo-spark`，因为我们要把预测结果直接保存到 MongoDB 里。和批处理模式不同的是，用 Spark Streaming 要同时做两件事（进行预测和把结果保存到 MongoDB 里）：

```
#!/usr/bin/env python

import sys, os, re
import json
import datetime, iso8601

from pyspark import SparkContext, SparkConf
from pyspark.sql import SparkSession, Row
from pyspark.streaming import StreamingContext
from pyspark.streaming.kafka import KafkaUtils, OffsetRange, TopicAndPartition

# 保存到 MongoDB 里
from bson import json_util
import pymongo_spark
pymongo_spark.activate()

def main(base_path):

  APP_NAME = “make_predictions_streaming.py"

  # Process data every 10 seconds
  PERIOD = 10
  BROKERS = 'localhost:9092'
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

```
PREDICTION_TOPIC = 'flight_delay_classification_request'

try:
  sc and ssc
except NameError as e:
  import findspark

  # 添加 streaming 包并初始化
  findspark.add_packages(
   ["org.apache.spark:spark-streaming-kafka-0-8_2.11:2.1.0"]
  )
  findspark.init()

  import pyspark
  import pyspark.sql
  import pyspark.streaming

  conf = SparkConf().set("spark.default.parallelism", 1)
  sc = SparkContext(
    appName="Agile Data Science: PySpark Streaming 'Hello, World!'", conf=conf
  )
  ssc = StreamingContext(sc, PERIOD)
  spark = pyspark.sql.SparkSession(sc).builder.appName(APP_NAME).getOrCreate()
```

加载模型的代码和 *ch08/make_predictions.py* 里的一模一样，就不再重复了。在 `main` 函数中，我们首先加载模型。Spark Streaming 的美在于可以复用批处理模式的 Spark 代码，这样你可以用批处理模式进行原型开发，或者为批处理系统和实时系统创建共用的代码库。

加载模型后，我们需要使用 `KafkaUtils.createDirectStream` 通过 Kafka 获取消息：

```
#
# 用 Streaming 处理预测请求
#

stream = KafkaUtils.createDirectStream(
  ssc,
  [PREDICTION_TOPIC],
  {
    "metadata.broker.list": BROKERS,
    "group.id": "0",
  }
)
```

因为消息是 JSON 格式的，所以我们需要进行解析。`pprint` 方法让我们可以在数据流经 Spark Streaming 时查看数据：

```
object_stream = stream.map(lambda x: json.loads(x[1]))
object_stream.pprint()
```


仅供非商业用途或交流学习使用

这时，我们的预测请求是 Python 的 dict 对象组成的 RDD。我们创建的模型是操作 DataFrame 的，因此我们要把 RDD 转为 DataFrame。为了实现转化，我们首先需要把 dict 对象转为 spark.sql.Row 对象。在转化中，我们需要使用 iso8601.parse_date（*https://pypi.python.org/pypi/iso8601*）把 ISO 8601 格式的日期字符串转化为 datetime 对象。和之前一样，我们使用 pprint 查看结果：

```
row_stream = object_stream.map(
  lambda x: Row(
    FlightDate=iso8601.parse_date(x['FlightDate']),
    Origin=x['Origin'],
    Distance=x['Distance'],
    DayOfMonth=x['DayOfMonth'],
    DayOfYear=x['DayOfYear'],
    UUID=x['UUID'],
    DepDelay=x['DepDelay'],
    DayOfWeek=x['DayOfWeek'],
    FlightNum=x['FlightNum'],
    Dest=x['Dest'],
    Timestamp=iso8601.parse_date(x['Timestamp']),
    Carrier=x['Carrier']
  )
)
row_stream.pprint()
```

下一步工作并不直观，因为所有的工作都在这一步发生。这似乎和 Spark 的数据流导向完全不沾边。然而，在 Spark Streaming 中你通常可以使用 DStream.foreachRDD（*http://bit.ly/2ouIxzW*）对 DStream 中的 RDD 进行一长串操作。从这一层意义上来说，Streaming 真的是位于 Spark 其他抽象之上的，让你可以把普通的 Spark 技术用于流数据。

我们从模型调用开始，只要一行代码：

```
# 用分类模型分析并把结果存入 MongoDB 中
row_stream.foreachRDD(classify_prediction_requests)
```

注意：我们把这个函数定义在 main 函数中，这样它就能访问我们在 main 函数中加载的模型。我们也可以把模型作为参数传递，但那样不优雅。在 main 函数中定义函数的缺点是外面别的脚本无法引入这些函数。

classify_prediction_requests 接收 RDD 作为参数，然后使用 SparkSession.createDataFrame 把我们准备好的 Row 对象转为完整的 DataFrame。和该脚本的批处理版本一样，我们需要首先配置表结构，这次多了一个 UUID 字段。一旦 DataFrame 创建出来，我们就能使用 DataFrame.show 查看情况了：

仅供非商业用途或交流学习使用

```
def classify_prediction_requests(rdd):

  from pyspark.sql.types import StringType, IntegerType, DoubleType, DateType,
    TimestampType
  from pyspark.sql.types import StructType, StructField

  prediction_request_schema = StructType([
    StructField("Carrier", StringType(), True),
    StructField("DayOfMonth", IntegerType(), True),
    StructField("DayOfWeek", IntegerType(), True),
    StructField("DayOfYear", IntegerType(), True),
    StructField("DepDelay", DoubleType(), True),
    StructField("Dest", StringType(), True),
    StructField("Distance", DoubleType(), True),
    StructField("FlightDate", DateType(), True),
    StructField("FlightNum", StringType(), True),
    StructField("Origin", StringType(), True),
    StructField("Timestamp", TimestampType(), True),
    StructField("UUID", StringType(), True),
  ])

  prediction_requests_df = spark.createDataFrame(
    rdd, schema=prediction_request_schema
  )
  prediction_requests_df.show()
```

和我们在批处理模式下所做的一样，我们需要根据 Origin 和 Dest 字段计算出 Route 字段：

```
#
# 添加 Route 变量取代 FlightNum
#

from pyspark.sql.functions import lit, concat
prediction_requests_with_route = prediction_requests_df.withColumn(
  'Route',
  concat(
    prediction_requests_df.Origin,
    lit('-'),
    prediction_requests_df.Dest
  )
)
prediction_requests_with_route.show(6)
```

现在我们有了 DataFrame，我们只要重复一遍 *ch08/make_preditions.py* 中的预测代码，预测结果也是 DataFrame 的形式。和批处理模式时一样，我们再用 show 查看这个输出：

```
# 使用列对应的管道向量化处理字符串字段
# 把类别字段转为分类特征向量，然后删掉中间字段
for column in ["Carrier", "DayOfMonth", "DayOfWeek", "DayOfYear",
               "Origin", "Dest", "Route"]:
  string_indexer_model = string_indexer_models[column]
```

```
  prediction_requests_with_route = string_indexer_model.transform(
    prediction_requests_with_route
  )

# 向量化处理数值列：DepDelay, Distance, and index columns
final_vectorized_features = vector_assembler.transform(
  prediction_requests_with_route
)

# 查看向量
final_vectorized_features.show()

# 删除索引值列
index_columns = ["Carrier_index", "DayOfMonth_index", "DayOfWeek_index",
                 "DayOfYear_index", "Origin_index", "Dest_index",
                 "Route_index"]
for column in index_columns:
  final_vectorized_features = final_vectorized_features.drop(column)

# 查看最终特征
final_vectorized_features.show()

# 进行预测
predictions = rfc.transform(final_vectorized_features)

# 删掉特征向量和预测元数据，以获取原始字段
predictions = predictions.drop("Features_vec")
final_predictions = predictions.drop("indices").drop("values") \
 .drop("rawPrediction").drop("probability")

# 查看输出
final_predictions.show()
```

最后，我们需要把预测结果 DataFrame 转回 RDD，因为 pymongo-spark 不支持 DataFrame，它支持 dict 对象组成的 RDD（甚至也不支持 pyspark.sql.Row 对象）。同样，如果 RDD 为空，调用 saveToMongoDB 也会导致崩溃，所以我们只在结果不为空时调用 saveToMongoDB：

```
# 保存到 Mongo
if final_predictions.count() >0:
  final_predictions.rdd.map(lambda x: x.asDict()).saveToMongoDB(
   "mongodb://localhost:27017/agile_data_science.flight_ \
     delay_classification_response"
  )
```

哟！搞定了。如果一切都按预期工作，我们的预测就会从 Kafka 传递到 Spark ML 再回到 MongoDB 中了。

测试整个系统

希望我们接下来可以尽情享受喜悦了！我们要端到端地测试整个系统。在开始这件事之前，我想花点时间回顾一下我们目前完成的事情。

系统整体总结

我们从网上搜集了商业航空公司 540 万条航班数据，通过探索性数据分析、交互式可视化以及搜索功能熟悉了数据。这让我们为使用 Spark 把航班数据转为统计模型的训练数据并预测航班延误的过程做好了准备，而 Spark 是可以处理任意规模的数据的。然后我们建立了一个网页前端，可以生成预测请求并发送到 Kafka 分布式队列中，而 Kafka 也能处理任意规模的数据。Spark Streaming 让我们利用批处理生成的模型，根据 Kafka 消息进行实时预测，将其结果存入 MongoDB 中，以便前端网页访问。最后，预测结果展示到了用户面前。

试水

好了，我们开始吧！从 bash 运行脚本来进行尝试：

```
python ch08/make_predictions_streaming.py .
```

输出量很大，但是关键部分是我们调用 pprint 和 show 所对应的输出。此时你应该看到如下所示的空输出（为适配页面而裁剪过）：

```
-------------------------------------------
Time: 2016-12-20 18:06:40
-------------------------------------------

+-------+----------+---------+---------+--------+
|Carrier|...|Timestamp|UUID|
+-------+----------+---------+---------+--------+

+-------+----------+---------+---------+--------+
|Carrier|...|NumericFeatures_vec|Features_vec|
+-------+----------+---------+---------+--------+

+-------+----------+---------+---------+--------+
|Carrier|...|UUID|Features_vec|
+-------+----------+---------+---------+--------+

+-------+----------+---------+---------+--------+
|Carrier|...|UUID|Prediction|
+-------+----------+---------+---------+--------+
```

现在，访问 *http://localhost:5000/flights/delays/predict_kafka*，同时打开 JavaScript 控制台玩一玩。输入一个非零的起飞延误、一个 ISO 格式的日期（我用的是 2016-12-25，在我写这

段内容的时候这是一个未来的时间）、一个有效的航空公司代码（如果你一个都不知道，可以使用 AA 或者 DL）、一组起降地（我最喜欢 ATL → SFO），以及一个有效的航班号（比如 1519），然后单击 Submit 按钮提交。观察 JavaScript 控制台中客户端向结果接口 */flights/delays/predict/classify_realtime/response/* 轮询数据的调试输出。

快速切回 Spark 控制台窗口。在 10 秒，也就是我们配置一个微型批处理的时长之内，你就能看到如下内容：

```
-------------------------------------------
Time: 2016-12-20 18:06:50
-------------------------------------------
{
 'Dest': 'ORD',
 'DayOfYear': 360,
 'FlightDate': '2016-12-25',
 'Distance': 606.0,
 'DayOfMonth': 25,
 'UUID': 'a01b5ccb-49f1-4c4d-af34-188c6ae0bbf0',
 'FlightNum': '2010',
 'Carrier': 'AA',
 'DepDelay': -100.0,
 'DayOfWeek': 6,
 'Timestamp': '2016-12-20T18:06:45.307114',
 'Origin': 'ATL'
 }

-------------------------------------------
Time: 2016-12-20 18:06:50
-------------------------------------------
Row(Carrier='AA', DayOfMonth=25, DayOfWeek=6, DayOfYear=360, DepDelay=-100.0,
Dest='ORD', Distance=606.0, FlightDate=datetime.datetime(2016,
12, 25, 0, 0, tzinfo=<iso8601.Utc>), FlightNum='2010',
Origin='ATL', Timestamp=datetime.datetime(2016, 12, 20, 18,
6, 45, 307114, tzinfo=<iso8601.Utc>),
UUID='a01b5ccb-49f1-4c4d-af34-188c6ae0bbf0')

+-------+----------+---------+---------+
|Carrier|...|               UUID|
+-------+----------+---------+---------+
|     AA|...|a01b5ccb-49f1-4c4...|
+-------+----------+---------+---------+

+-------+----------+---------+---------+
|Carrier|...|        Features_vec|
+-------+----------+---------+---------+
|     AA|...|(8009,[2,38,51,34...|
+-------+----------+---------+---------+

+-------+----------+---------+---------+
|Carrier|...|        Features_vec|
+-------+----------+---------+---------+
```

```
|      AA|...|(8009,[2,38,51,34...|
+-------+----------+---------+---------+

+-------+----------+---------+---------+
|Carrier|...|Prediction |
+-------+----------+---------+---------+
|      AA|...|        0.0|
+-------+----------+---------+---------+
```

举一反三

回到浏览器，预测结果正骄傲地呈现在页面上，见图 8-11。看起来好厉害的样子，对吧？在我写这段内容的时候，我一遍一遍地提交请求，同时惊叹于各模块的完美融合，以及同样的这份代码可以利用大型 Spark 集群和相匹配的数百台网络服务器支持 PB 级的数据的情形。我们在本书中解决的一切疑难都是为了构建出可以伸展到那个级别的应用程序。我希望本书和本书所描述的这个应用能作为一个示范，让你从中学习并衍生出自己的应用。

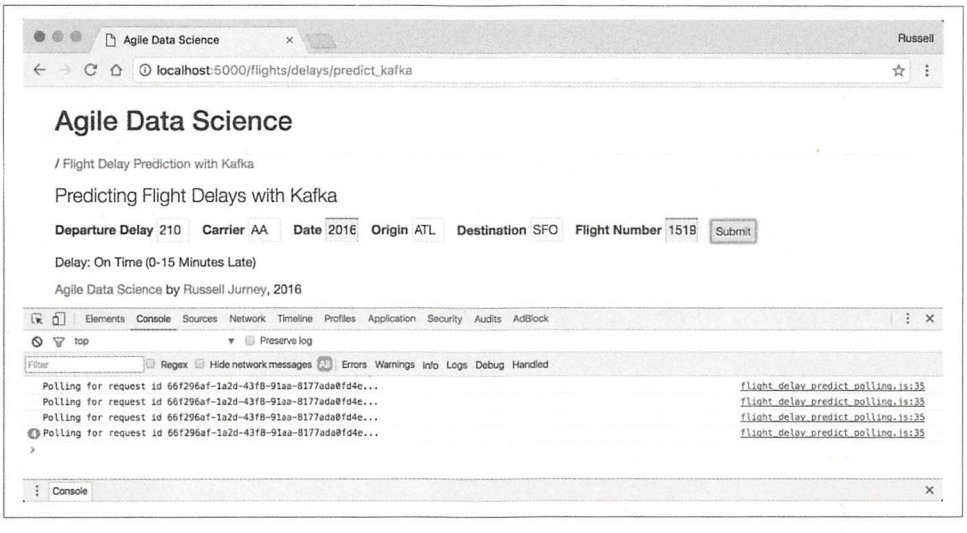


图 8-11　在 Kafka 和 Spark Streaming 处理预测请求时轮询航班延误预测结果

本章小结

在本章中，我们把第 7 章所创建的预测模型整合到了基于网页的真实产品中。我们使用了三种不同的方式：一是使用 `scikit-learn` 实现实时的网页服务，二是使用 Spark 和

Airflow 批量更新的服务，三是使用 Kafka 和 Spark Streaming 的次实时级服务。我希望你能使用我们在本章中介绍的知识把自己的模型部署到新型数据产品完整的预测系统中。

我会很高兴了解你正在怎样使用敏捷数据科学以及本章中的示例来构建你自己的产品，我也很愿意提供帮助。不要犹豫，联系我吧。你可以发邮件给我的个人邮箱 *russell.jurney@gmail.com*，或是注册敏捷数据科学的邮件列表 *agile-data-science@googlegroups.com* 参与讨论。

现在我们转为改进之前创建的模型。一旦一个模型以一种产品的形式面世并逐渐流行，产品开发就常常变成一种无情的驱动促使模型不断改善，这会成为一种基本要求。我希望你们的产品有这个“问题”，这也是我们接下来的重点。

第 9 章

改进预测结果

现在我们已经部署了预测航班延误的模型，是时候根据用户反馈证明我们的预测是有用的了。要想预测足够有价值，预测质量非常重要。这样一来，也是时候迭代改进预测质量了。如果预测足够有价值，那么这会成为一个人或多个人的全职工作。

在本章中，我们将对 Spark ML 分类器进行调优，并进行额外的特征工程以提高预测质量。通过这些工作，我们将展示如何迭代改进预测。

本章的代码示例在 *Agile_Data_Code_2/ch09* 中。克隆代码仓库，跟着做吧！

```
git clone https://github.com/rjurney/Agile_Data_Code_2.git
```

解决预测的问题

写到此处，我们发现，我们的模型不论输入是什么，始终会预测出同一种结果。我们从使用 *ch09/Debugging Prediction Problems.ipynb* 的 Jupyter 笔记本开始进行调查。

笔记本本身很长，我们进行了很多尝试来修复模型。结果是我们犯了错误。在编码称名 / 类别字符串特征时，我们对 `StringIndexerModel` 的输出使用了 `OneHotEncoder`。这是为除决策树以外的模型进行特征编码的方式。如果是决策树模型的话，你应该直接拿 `StringIndexerModel` 给出的字符串索引值去用 `VectorAssembler` 和连续 / 数值特征进行组合。决策树可以推断出索引值是代表分类的。直接把字符串索引值放入特征向量的一大好

处是你能获得易于解读的特征重要数据。

当我们发现这个问题时，我们只好回过头去修改前面的内容，这样我们才不至于教给你一些错误的东西，因此你看到的是已经修改后的内容。不过我们觉得展示该笔记本中的这个过程是有意义的，这展现了在没有指导的情况下要怎样解决问题：构造出不能用的破玩意儿，然后把它修好。

什么时候需要改进预测

不是所有的预测都需要改进。通常情况下，“糙快猛”的东西足以用作 MVP（最小可行产品）。只有被证明为有用的预测模型才需要改进。为了提升预测质量，可能会浪费大量时间，因此在投入改进预测的任务之前，应该先和用户进行交流。这也是我们为什么要把改进预测的讨论单独作为一章的原因。

改进预测表现

有这样一些方法可以改进已有的预测模型。第一种是对统计模型进行参数调优。第二种是采用特征工程。

为改进预测模型质量，可以对模型超参数进行调优，这可以凭感觉完成，也可以使用网格搜索或随机搜索暴力完成。我们这里关注的是特征工程，因为超参数调优在别处讲过。在 Spark 关于模型选择和调优的文档（*https://spark.apache.org/docs/latest/ml-tuning.html*）中包含一份不错的超参数调优指导手册。

随着这一章的展开，我们将利用到目前为止已经完成的工作来进行特征工程。特征工程是进行优秀预测最为重要的部分。它使用通过探索性数据分析发现的数据相关特性，为机器学习算法提供更好、更重要的数据作为输入。

黏附试验法：找出黏性好的

有多种方法来决定使用哪些特征，Saurav Kaushik 发表在 Analytics Vidhya 上的文章对这些方法介绍得挺好。我们给主要使用的方法起了一个有趣的名字，叫*黏附试验法*（*Experimental Adhesion Method*），它快速选择所有能轻易计算得到的特征，使用随机森林或者渐进梯度决策树模型来尝试使用全部特征（注意即使应用程序需要另一种模型，我们还是使用决策

仅供非商业用途或交流学习使用

树来辅助特征选择)。然后我们训练模型并检查模型的特征重要性,看看哪些特征“黏性好”,保留最重要的一些变量,这样就构成了我们起步用的基础模型。

特征工程是一个迭代的过程。我们基于特征重要性,思考哪些数据可以尝试引入为新的特征。我们从最简单的想法或者最容易实现的想法开始。如果特征重要性表明一种特征重要,而我们无法轻易计算出与之类似的新特征,那么我们可以考虑如何获取新数据,与训练数据连接,作为特征使用。

关键是在探索特征空间时保持逻辑性与系统性。你应该考虑计算潜在特征的难易程度,以及如果该特征成为重要特征的话我们能学到什么。有没有其他相似的能拿来试一试的备选特征?提出假设并以新特征的形式进行测试。在试验中评估每个新特征,并在对下一个特征开展工作前反思学到的经验。

为试验建立严格的指标

为了改进我们的分类模型,首先需要能可靠地衡量预测质量。因此,我们要加强交叉验证的代码,然后用初始模型跑出模型质量的基线。如 *ch09/baseline_spark_mllib_model.py* 所示,这是我们从 *ch09/train_spark_mllib_model.py* 中复制过来的,并对交叉验证代码进行了改进。

为了评估分类器的预测质量,需要使用不止一项的指标。Spark ML 的 `MulticlassClassificationEvaluator` 提供了四项指标:准确率、加权精确率、加权召回率、*f1* 值。

定义分类模型的各项指标

准确率(*accuracy*),顾名思义,是指正确的预测数除以预测总数的结果。这是我们首先检查的指标,但是单看这个还不够。*精确率*(*precision*)衡量结果多么有用。*召回率*(*recall*)表示结果的完整性。*f1 值*则结合了精确率和召回率来决定整体质量。总而言之,在连续几次训练模型之间,这些指标发生的变化可以让我们清楚地看到模型预测质量的改变。我们要使用这些指标和特征重要性一同引导特征工程工作的方向。

仅供非商业用途或交流学习使用

特征重要性

模型质量指标不足以引导模型的迭代改进。要理解每次新的运行发生了什么，我们需要使用一种名为*决策树*（*https://en.wikipedia.org/wiki/Decision_tree*）的模型。

在 Spark ML 中，支持多元分类的最佳通用分类模型是*随机森林*的一个实现 `RandomForestClassificationModel`，通过 `RandomForestClassifier` 拟合，随机森林可以用于分类或回归，还有一个重要功能可以帮助我们评估单个特征对预测模型的影响，这就是*特征重要性*（*feature importance*）。

顾名思义，所谓特征的重要性就是衡量一个特征对模型的准确率有多重要。这一信息极其重要，因为它可以作为*特征工程*的指导方针。换句话说，如果你知道一个特征有多重要，你就可以使用这一线索来做出提高模型准确率的改变，比如删除不重要的特征，尝试发掘出和已知很重要的特征相似的特征。特征工程是敏捷数据科学的一个重大主题，这也是我们之前进行迭代可视化和数据探索的重要原因（目的在于把数据搞明白，驱动特征工程开展工作）。

注意：在许多分类和回归任务中，最先进的技术是*渐进梯度决策树*，但是就 2.1.0 版本的 Spark ML 实现而言，该算法对应的用 `GBTClassifier` 拟合出的 `GBTClassificationModel` 仅能用于二元分类问题。

实施严格的试验

为了信任试验得出的各项指标，我们需要把试验重复至少两次，观察试验之间的差距。另外，我们需要循环运行每个指标的测量代码。一旦我们每个指标都收集到了几个分数，我们就可计算出每个指标分数的均值和标准差。总之，这些分数让我们对分类器的质量有了宏观的认识。

首先，我们需要迭代并重复试验 *N* 次。每次试验我们都要划分一次测试 / 训练数据集，然后使用训练数据来训练模型，用测试数据进行测试。接着我们使用 `MulticlassClassificationEvaluator` 获取各项指标的分数。我们把每个指标的分数分别收集到一个列表中，试验的最后我们将进行评估：

```
#
# 分类器交叉验证、训练、评估：循环 3 次计算 4 项指标
#

from collections import defaultdict
```

仅供非商业用途或交流学习使用

```
scores = defaultdict(list)
metric_names = ["accuracy", "weightedPrecision", "weightedRecall", "f1"]
split_count = 3

for i in range(1, split_count + 1):
  print("\nRun {} out of {} of test/train splits in cross validation...". \
    format(
      i,
      split_count,
    )
  )

  # 测试 / 训练划分好的数据
  training_data, test_data = final_vectorized_features.limit(1000000).\
    randomSplit([0.8, 0.2])

  # 实例化随机森林分类器，并使用全部训练数据进行拟合
  from pyspark.ml.classification import RandomForestClassifier
  rfc = RandomForestClassifier(
    featuresCol="Features_vec",
    labelCol="ArrDelayBucket",
    predictionCol="Prediction",
    maxBins=4657
  )
  model = rfc.fit(training_data)

  # 保存新模型，覆盖旧模型
  model_output_path = " \
    {}/models/spark_random_forest_classifier.flight_delays.baseline.\
      bin".format(
    base_path
  )
  model.write().overwrite().save(model_output_path)

  # 使用测试数据评估模型
  predictions = model.transform(test_data)

  # 评估这种数据集划分下各项指标的结果
  from pyspark.ml.evaluation import MulticlassClassificationEvaluator
  for metric_name in metric_names:
    evaluator = MulticlassClassificationEvaluator(
      labelCol="ArrDelayBucket",
      predictionCol="Prediction",
      metricName=metric_name
    )
    score = evaluator.evaluate(predictions)

    scores[metric_name].append(score)
    print("{} = {}".format(metric_name, score))
```

仅供非商业用途或交流学习使用

这样我们就获得了保存着分数的 defaultdict 对象，每项指标对应一个分数列表。现在我们需要计算各列表的均值和标准差，这样就得到了每项指标的整体均值和标准差：

```
#
# 计算每项指标的均值和标准差，输出一张表格
#
import numpy as np
score_averages = defaultdict(float)

# 计算表格数据
average_stds = []
for metric_name in metric_names:
  metric_scores = scores[metric_name]

  average_accuracy = sum(metric_scores) / len(metric_scores)
  score_averages[metric_name] = average_accuracy

  std_accuracy = np.std(metric_scores)

  average_stds.append((metric_name, average_accuracy, std_accuracy))

# 输出表格
print("\nExperiment Log")
print("--------------")
print(tabulate(average_stds, headers=["Metric", "Average", "STD"]))
```

结果如下：

```
Experiment Log
--------------
Metric                Average          STD
-----------------  ---------  -----------
accuracy            0.594443  0.000382382
weightedPrecision   0.642419  0.00352101
weightedRecall      0.594443  0.000382382
f1                  0.522397  0.000438121
```

这些标准差很小，表明我们甚至可以不用进行 k 折交叉验证，但是查看实际分数却会得出相反的结论：

```
$ scores
defaultdict(list,
            {'accuracy': [
              0.5960317877085193,
              0.5962539640360968,
              0.5962346664334288
              ],
             'f1': [0.5251883509444727,
                    0.5266212073123311,
                    0.5258877000496558],
             'weightedPrecision': [0.6495952645815938,
```

仅供非商业用途或交流学习使用

```
                                0.6498757953978488,
                                0.6549703272382899],
'weightedRecall': [0.5960317877085194,
                   0.5962539640360968,
                   0.5962346664334288]})
```

实际上每次循环之间的差距还是很明显的，这可能会掩盖预测质量小幅的提高（或者下降）。

这样的迭代很费时间，也不利于实验。我们要找一个折中的方案。

通过实验比较判断是否提高

我们已经有了指标基准线，可以在改进模型时重新运行代码，查看计算的这四个指标的变化情况。看起来准备工作都完成了，我们可以开始处理模型和特征了。不过，我们很快就会遇到问题。我们会丢失先前运行的分数变化轨迹，因为分数会被每次运行脚本时在屏幕上输出的大量日志挤掉。除非我们每次都把结果抄下来，但这也太愚蠢了。因此，我们要把这个过程自动化起来。

我们要做的是从磁盘上读取分数日志，按照之前的方式评估当前的分数，在日志中添加一条新记录，把当前的分数存回日志以便下次运行时访问。实现这样功能的代码如下。

首先我们使用 `pickle` 加载已有的分数记录。如果没有任何记录，我们就初始化一个新的日志，也就是 Python 中一个空的 `list` 对象。接下来，我们准备一条新的日志记录——就是一个包含四项指标均值的 Python 的 `dict` 对象。然后我们从这次运行的分数中减掉上一轮的分数，求出本次运行的变化值。我们用这个差来评估改变是否有用（要和特征重要性的变化一起看，稍后会讲到）。最后，我们把新的分数追加到日志后面，写回磁盘：

```
#
# 把分数保存到运行后不清空的分数日志里
#
import pickle

# 读取分数日志，或者初始化空日志
try:
  score_log_filename = "{}/models/score_log.pickle".format(base_path)
  score_log = pickle.load(open(score_log_filename, "rb"))
  if not isinstance(score_log, list):
    score_log = []
except IOError:
  score_log = []

# 计算本次分数对应的日志条目
score_log_entry = {
  metric_name: score_averages[metric_name] for metric_name in metric_names
```

仅供非商业用途或交流学习使用

```
}

# 计算并展示各项指标分数的变化
try:
  last_log = score_log[-1]
except (IndexError, TypeError, AttributeError):
  last_log = score_log_entry

experiment_report = []
for metric_name in metric_names:
  run_delta = score_log_entry[metric_name] - last_log[metric_name]
  experiment_report.append((metric_name, run_delta))

print("\nExperiment Report")
print("-----------------")
print(tabulate(experiment_report, headers=["Metric", "Score"]))

# 把本次的均分追加到日志里
score_log.append(score_log_entry)

# 保存日志文件以备后用
pickle.dump(score_log, open(score_log_filename, "wb"))
```

现在当我们运行脚本时，便会获得与上轮运行脚本比较的变化报告。我们可以利用两轮运行之间的变化和特征重要性来指导改进模型的方向。例如，在某轮测试运行中模型准确率提高了 0.003 ：

```
Experiment Report
-----------------
Metric                    Score
-----------------  -----------
accuracy            0.00300548
weightedPrecision  -0.00592227
weightedRecall      0.00300548
f1                 -0.0105553
```

检查特征重要性的变化

我们可以使用传给 `VectorAssembler` 的那些列，以及 `RandomForestClassificationModel.featureImportances` 来获取每个具名特征的特征重要性。这些重要性数值很有价值，因为它们和我们的预测质量分数一样，可以让我们看出每轮迭代之间所有特征的特征重要性变化情况。如果发现新引入的特征还算重要，那么只要该特征没有导致预测质量下降，我们就值得把该特征加入模型中。

我们首先修改实验迭代，记录每轮实验的特征重要性。参考下面的代码，它是 *ch09/improved_spark_mllib_model.py* 的缩略版本 ：

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

```
feature_importances = defaultdict(list)

...

for i in range(1, split_count + 1):
print("\nRun {} out of {} of test/train splits in cross validation...".format(
      i,
      split_count,
    )
  )

...

#
# 收集特征重要性
#
feature_names = vector_assembler.getInputCols()
feature_importance_list = model.featureImportances
for feature_name, feature_importance in \
    zip(feature_names, feature_importance_list):
  feature_importances[feature_name].append(feature_importance)
```

接下来，我们需要计算每个特征的特征重要性的均值。注意：我使用 defaultdict（float）来确保在访问不存在的键时也会返回 0。这在比较两组不同特征产生的日志时有重要的作用：

```
# 计算每个特征的重要性平均值
feature_importance_entry = defaultdict(float)
for feature_name, value_list in feature_importances.items():
  average_importance = sum(value_list) / len(value_list)
  feature_importance_entry[feature_name] = average_importance
```

为了输出特征重要性，我们需要把数据按照重要性进行降序排序：

```
# 将特征重要性降序排序并输出
import operator
sorted_feature_importances = sorted(
  feature_importance_entry.items(),
  key=operator.itemgetter(1),
  reverse=True
)

print("\nFeature Importances")
print("-------------------")
print(tabulate(sorted_feature_importances, headers=['Name', 'Importance']))
```

接下来，我们需要进行和模型分数日志一样的工作：读取模型，为本轮实验创建一条记录，读取上轮实验结果并计算与本轮实验相比各特征的重要性变化情况，然后输出差值报告。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

首先，我们读取上一轮的特征日志。如果日志不存在而无法读到的话，我们就把上一轮特征日志 last_feature_log 的所有特征都初始化为 0，这样新的特征总会得到和实际值相等的正数分数：

```
# 读取特征重要性日志，或者初始化空日志
try:
  feature_log_filename = "{}/models/feature_log.pickle".format(base_path)
  feature_log = pickle.load(open(feature_log_filename, "rb"))
if not isinstance(feature_log, list):
    feature_log = []
except IOError:
  feature_log = []

# 计算并显示每个特征的得分变化情况
try:
  last_feature_log = feature_log[-1]
except (IndexError, TypeError, AttributeError):
  last_feature_log = defaultdict(float)
  for feature_name, importance in feature_importance_entry.items():
    last_feature_log[feature_name] = importance
```

接下来，我们计算上一次和这次运行的测试之间的差距：

```
# 计算差值
feature_deltas = {}
for feature_name in feature_importances.keys():
  run_delta = \
    feature_importance_entry[feature_name] - last_feature_log[feature_name]
  feature_deltas[feature_name] = run_delta
```

为了展示这些差值，我们进行了降序排序，把差值最大的放在最前面：

```
# Sort feature deltas, biggest change first
import operator
sorted_feature_deltas = sorted(
  feature_deltas.items(),
  key=operator.itemgetter(1),
  reversed=True
)
```

然后，展示排好序的特征差值列表：

```
# 展示排好序的特征差值
print("\nFeature Importance Delta Report")
print("-------------------------------")
print(tabulate(sorted_feature_deltas, headers=["Feature", "Delta"]))
```

最后，和预测质量的分数日志一样，我们把本轮运行的记录追加到日志文件中，写回磁盘，以备下一轮使用：

```
# 把本轮的平均差值追加到日志中
feature_log.append(feature_importance_entry)
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

```
# 保存日志准备下一次运行
pickle.dump(feature_log, open(feature_log_filename, "wb"))
```

首次测试模型会生成如下的输出。我们既要使用原始的特征重要性结果，也要使用特征重要性变动情况，来指导改进模型时特征的创建或者替换的工作：

```
 Experiment Log
--------------
Metric                Average          STD
-----------------  ---------  -----------
accuracy            0.594014  0.000270987
weightedPrecision   0.570674  0.0821537
weightedRecall      0.594014  0.000270987
f1                  0.521789  3.70999e-05

Experiment Report
-----------------
MetricScore
-----------------  ------------
accuracy           -0.000429286
weightedPrecision  -0.0717445
weightedRecall     -0.000429286
f1                 -0.000608931

Feature Importances
-------------------
Name                Importance
---------------  ------------
DepDelay          0.882216
Route_index       0.0571401
Origin_index      0.0142741
Distance          0.0134583
Dest_index        0.00745796
DayOfYear         0.00544761
Carrier_index     0.00454088
DayOfMonth        9.31109e-05
DayOfWeek         5.2597e-05

Feature Importance Delta Report
-------------------------------
Feature             Delta
---------------  -------
Distance               0
Dest_index             0
DayOfWeek              0
Origin_index           0
DayOfYear              0
Carrier_index          0
Route_index            0
DayOfMonth             0
DepDelay               0
```


仅供非商业用途或交流学习使用

小结

既然我们已经有能力理解实验迭代之间的变化了，那么就可以检测出哪些改动对模型起到改进作用。我们可以先添加特征，测试特征对模型预测质量的作用，然后寻找相关的特征来提高质量！如果没有这部分基础，我们很难做出正面的改进。有了它，我们唯一的障碍就是在改进模型时的创新能力了。

把当日时间作为特征

在查看特征重要性报告时，日期 / 时间字段似乎是有一些作用的。如果我们把起飞 / 到达字段中的时 / 分以整型提取出来会怎么样？这能让我们的模型区分上午的航班、下午的航班以及红眼航班，这显然对准点情况有影响，因为上午的空域流量比半夜里的要大。

参见 *ch09/explore_delays.py*。让我们先探索该特征的假设是否成立，也就是航班延误情况是否与当日时间有关：

```
spark.sql("""
  SELECT
    HOUR(CRSDepTime) + 1 AS Hour,
    AVG(ArrDelay),
    STD(ArrDelay)
  FROM features
  GROUP BY HOUR(CRSDepTime)
  ORDER BY HOUR(CRSDepTime)
""").show(24)
```

结果如下所示：

```
+----+-------------------+---------------------+
|Hour|      avg(ArrDelay)|stddev_samp(ArrDelay)|
+----+-------------------+---------------------+
|   1|-0.9888343067527446|    35.96846550716142|
|   2|0.21487576223466862|   35.744333727508334|
|   3| 1.5671059921857282|    35.00946190001324|
|   4| 2.3711289989006086|   36.182339627895345|
|   5| 3.0942288270090894|   37.547244850760876|
|   6|  4.239319300385845|   38.400571868893834|
|   7|  5.234954994309625|    39.28255300783613|
|   8|  6.453546045667625|    39.99971120960918|
|   9|  7.186654216429772|    41.40488311224806|
|  10|  8.365290552625943|   42.940647757026625|
|  11|  9.268328745619563|   43.626137917652855|
|  12|  9.841703616195401|    43.52976518121594|
|  13| 10.066688650580275|    41.92576203774942|
|  14|  9.283710900023337|     40.6576680093127|
|  15|  7.423578894503908|    37.93024949987321|
|  16|  6.026947232249046|     36.2827909463706|
```


仅供非商业用途或交流学习使用

```
|  17|  2.878606342393896|   34.521580465809635|
|  18|  1.202488132263873|   35.281643789718856|
|  19|  3.921360847741216|    51.57255339085103|
|  20|  1.416023166023166|    35.07002923779163|
|  21|   1.01067615658363|   33.710428616724336|
|  22|-1.6537734227264913|    44.14071722078961|
|  23|-2.4204632317424886|    38.33508514261801|
|  24|-2.3249719752460805|     35.6965483893959|
+----+-------------------+---------------------+
```

航班的原定时间确实有影响！定于下午一点起飞的航班平均延误 10 分钟，而相比之下晚上十一点起飞的航班则平均提前 2.5 分钟到达目的地。标准差则变化不大，不过最高值还是在中午附近出现。看起来是个值得纳入的特征！

既然讲到这里了，原定的到达时间又如何呢？我们也拿 CRSArrTime 进行同样的计算：

```
spark.sql("""
  SELECT
    HOUR(CRSArrTime) + 1 AS Hour,
    AVG(ArrDelay),
    STD(ArrDelay)
  FROM features
  GROUP BY HOUR(CRSArrTime)
  ORDER BY HOUR(CRSArrTime)
""").show(24)
```

结果如下：

```
+----+--------------------+---------------------+
|Hour|       avg(ArrDelay)|stddev_samp(ArrDelay)|
+----+--------------------+---------------------+
|   1| -1.7116259174208655|    36.33240606655376|
|   2| -1.2394161336909428|    34.65716885698246|
|   3|  -0.560109126391461|    35.93678468759135|
|   4|-0.03119026777898...|    34.18894939261768|
|   5|  1.0004041388403222|    34.89927883531852|
|   6|  1.8046307093420586|   35.983884598879854|
|   7|  2.7098903974183797|   36.828717160294616|
|   8|  3.2653490352035015|    37.94922697845916|
|   9|   4.460970473403804|    38.75981742307256|
|  10|   5.733407037370677|   39.332218073928395|
|  11|   7.415162373324524|     41.6390996526558|
|  12|   8.394327378488986|   42.304599584222764|
|  13|    9.13641026800476|    44.15236003931785|
|  14|   9.263586544185449|    43.95699577126197|
|  15|   9.463244251854364|   42.694962099183385|
|  16|   9.158153249212814|   40.631824365179185|
|  17|   8.851837125560714|    39.32989266521008|
|  18|   8.374134395914735|    40.76013328408966|
|  19|   6.383113511268045|   40.175828363537185|
|  20|   4.743589743589744|   33.043381854132626|
```


仅供非商业用途或交流学习使用

```
|  21|   6.129032258064516|   43.144599976836396|
|  22|  6.6219806017174276|    41.76914003060987|
|  23|  0.9266386975097186|   41.750524061849795|
|  24| -1.0140736298134196|    40.40944633965604|
+----+--------------------+---------------------+
```

结果相似，同样很重要。我们也要把这个特征加上。

让我们基于 *ch09/train_spark_mllib_model.py* 为改进后的新模型新建一个文件，如 *ch09/improved_spark_mllib_model.py* 所示。添加 CRSDepHourOfDay 和 CRSArrHourOfDay 列的代码很简单：

```
from pyspark.sql.functions import hour
features_with_hour = features_with_route.withColumn(
  "CRSDepHourOfDay",
  hour(features.CRSDepTime)
)
features_with_hour = features_with_hour.withColumn(
  "CRSArrHourOfDay",
  hour(features.CRSArrTime)
)
features_with_hour.select(
  "CRSDepTime",
  "CRSDepHourOfDay",
  "CRSArrTime",
  "CRSArrHourOfDay").show()
```

结果如下：

```
+--------------------+---------------+--------------------+---------------+
|          CRSDepTime|CRSDepHourOfDay|          CRSArrTime|CRSArrHourOfDay|
+--------------------+---------------+--------------------+---------------+
|2015-01-01 07:30:...|              7|2015-01-01 10:10:...|             10|
|2014-12-31 23:25:...|             23|2015-01-01 02:15:...|              2|
|2015-01-01 01:00:...|              1|2015-01-01 03:45:...|              3|
|2015-01-01 09:55:...|              9|2015-01-01 11:30:...|             11|
|2015-01-01 00:55:...|              0|2015-01-01 02:25:...|              2|
|2015-01-01 05:45:...|              5|2015-01-01 07:15:...|              7|
|2015-01-01 02:45:...|              2|2015-01-01 04:15:...|              4|
|2015-01-01 07:25:...|              7|2015-01-01 08:50:...|              8|
|2015-01-01 11:00:...|             11|2015-01-01 12:30:...|             12|
|2015-01-01 12:15:...|             12|2015-01-01 13:40:...|             13|
|2015-01-01 03:55:...|              3|2015-01-01 05:25:...|              5|
|2015-01-01 08:40:...|              8|2015-01-01 10:05:...|             10|
|2015-01-01 00:15:...|              0|2015-01-01 02:12:...|              2|
|2014-12-31 23:00:...|             23|2015-01-01 00:52:...|              0|
|2015-01-01 13:10:...|             13|2015-01-01 15:02:...|             15|
|2015-01-01 05:30:...|              5|2015-01-01 06:35:...|              6|
|2014-12-31 21:50:...|             21|2014-12-31 22:50:...|             22|
|2015-01-01 00:30:...|              0|2015-01-01 01:40:...|              1|
|2015-01-01 01:05:...|              1|2015-01-01 02:15:...|              2|
|2015-01-01 07:55:...|              7|2015-01-01 08:55:...|              8|
+--------------------+---------------+--------------------+---------------+
```


仅供非商业用途或交流学习使用

接下来的代码把列数据转化为索引值，并把处理好的列包括到最终的 Feature_vec 中。我们会跳过这一部分，你可以在 *ch09/improved_spark_mllib_model.py* 中找到相应的代码。由于我们已经配置好了用来测试与比较两轮迭代之间预测质量改变的实验代码，所以可以通过 bash 直接运行测试脚本：

```
python ch09/improved_spark_mllib_model.py .
```

输出结果如下：

```
Experiment Log
--------------
Metric                 Average          STD
-----------------  ---------  -----------
accuracy            0.594656  0.000509343
weightedPrecision   0.641538  0.00372632
weightedRecall      0.594656  0.000509343
f1                  0.5233    0.000700844

Experiment Report
-----------------
Metric                   Score
-----------------  ----------
accuracy           0.00108926
weightedPrecision  0.0154773
weightedRecall     0.00108926
f1                 0.00210414

Feature Importances
-------------------
Name                Importance
---------------  ------------
DepDelay           0.886486
Route_index        0.0598883
Distance           0.0129897
Origin_index       0.0120841
CRSArrHourOfDay    0.00982592
Dest_index         0.00877569
Carrier_index      0.00448823
DayOfYear          0.00412094
CRSDepHourOfDay    0.0012717
DayOfWeek          6.92304e-05
DayOfMonth         2.07392e-07

Feature Importance Delta Report
-------------------------------
Feature                  Delta
---------------  ------------
CRSArrHourOfDay    0.00982592
```


仅供非商业用途或交流学习使用

```
DepDelay            0.00671702
CRSDepHourOfDay     0.0012717
DayOfWeek          -2.72792e-05
DayOfMonth         -8.81635e-05
Origin_index       -0.00152788
Distance           -0.00188322
Route_index        -0.00214609
Dest_index         -0.0032327
DayOfYear          -0.00377184
Carrier_index      -0.00513747
```

解释一下这个输出，这两个字段的特征重要性合起来约为 1%，不过对模型准确性的作用不大。我们会保留这些字段，尽管没起很大作用。在不采用先进的时间序列分析的前提下，似乎我们已经把基于时间戳的特征压榨干净了。

纳入飞机数据

回顾第 162 页“调查飞机（实体）”，那时我们把飞机的制造商数据纳入了数据模型中。当时，我们分析了各制造商在美国商业机队中的占有率。在本节中，我们要把这些数据和航空公司数据进行连接，查看对模型的准确率会有怎样的影响。

我想知道飞行器（也就是航班的“金属”）的属性是否会影响延误情况？比如，大飞机飞得更高，相比起小飞机，受航路天气影响小一些。我实在想不出引擎制造商、飞机制造商、制造年份这些因素怎么会影响预测模型的结果，不过既然我们在引入一个新字段，不妨尝试所有这些特征！注意对于影响不大的特征，我们可以直接丢弃。我们的决策树实验模型的美妙之处就在于，引入多个字段时不会产生多余开销。有时候你可以让模型自己选择重要的特征。

注意：团队成员或其他团队有时要占用时间来协调工作，和他们打交道时，恰当描述你正在尝试的实验可以让团队保持同步。例如，“我们正在尝试把一个从 FAA 网站爬取的数据集纳入我们的航班延误预测模型里”就是敏捷冲刺中一个很好的实验描述。

提取飞机特征

要往模型中添加飞机特征，我们需要先创建一个新的特征提取脚本 *ch09/extract_features_with_airplanes.py*。可以从 *ch09/extract_features.py* 中复制一份出来进行修改得到这个脚本。我们将略过和原文件重复的部分代码（第 7 章中已经讲解过），仅仅展示有修改的部分。

仅供非商业用途或交流学习使用

首先，从训练数据中取数据时我们多取了 TailNum 字段。因为这个字段同样存在于飞机数据集中，我们需要把它改个名字，要不然在连接之后访问就不太容易了。我们把它命名为 FeatureTailNum：

```
# 选取一些有意义的特征
simple_on_time_features = spark.sql("""
SELECT
  FlightNum,
  FlightDate,
  DayOfWeek,
  DayofMonth AS DayOfMonth,
  CONCAT(Month, '-',  DayofMonth) AS DayOfYear,
  Carrier,
  Origin,
  Dest,
  Distance,
  DepDelay,
  ArrDelay,
  CRSDepTime,
  CRSArrTime,
  CONCAT(Origin, '-', Dest) AS Route,
  TailNum AS FeatureTailNum
FROM on_time_performance
""")

simple_on_time_features.select(
  "FlightNum",
  "FlightDate",
  "FeatureTailNum"
).show(10)

...

def alter_feature_datetimes(row):

  flight_date = iso8601.parse_date(row['FlightDate'])
 scheduled_dep_time = convert_datetime(row['FlightDate'], row['CRSDepTime'])
 scheduled_arr_time = convert_datetime(row['FlightDate'], row['CRSArrTime'])

  # 处理过夜航班
  if scheduled_arr_time <scheduled_dep_time:
    scheduled_arr_time += datetime.timedelta(days=1)

  doy = day_of_year(row['FlightDate'])

  return {
    'FlightNum': row['FlightNum'],
    'FlightDate': flight_date,
    'DayOfWeek': int(row['DayOfWeek']),
    'DayOfMonth': int(row['DayOfMonth']),
    'DayOfYear': doy,
    'Carrier': row['Carrier'],
```

仅供非商业用途或交流学习使用

```
    'Origin': row['Origin'],
    'Dest': row['Dest'],
    'Distance': row['Distance'],
    'DepDelay': row['DepDelay'],
    'ArrDelay': row['ArrDelay'],
    'CRSDepTime': scheduled_dep_time,
    'CRSArrTime': scheduled_arr_time,
    'Route': row['Route'],
    'FeatureTailNum': row['FeatureTailNum'],
  }
timestamp_features = filled_on_time_features.rdd.map(alter_feature_datetimes)
timestamp_df = timestamp_features.toDF()
```

接下来，我们读取飞机数据，用特征数据集左连接它。注意：空值会导致 StringIndexer 出现问题。但是我们也不想丢弃空值或者有空值的整行数据，因为决策树模型不能学习一个值是否存在。我们使用 DataFrame.selectExpr，把空值和字符串 'Empty' 进行 COALESCE。这样空值就可以从 StringIndexer 中获得自己的索引值，程序就能正常工作了。还要注意的是，我们把 FeatureTailNum 重命名回 TailNum，以便最后输出时使用：

```
# 读取飞机数据，按机尾编号进行左连接
airplanes_path = "{}/data/airplanes.json".format(
  base_path
)
airplanes = spark.read.json(airplanes_path)

features_with_airplanes = timestamp_df.join(
  airplanes,
  on=timestamp_df.FeatureTailNum == airplanes.TailNum,
  how="left_outer"
)

features_with_airplanes = features_with_airplanes.selectExpr(
  "FlightNum",
  "FlightDate",
  "DayOfWeek",
  "DayOfMonth",
  "DayOfYear",
  "Carrier",
  "Origin",
  "Dest",
  "Distance",
  "DepDelay",
  "ArrDelay",
  "CRSDepTime",
  "CRSArrTime",
  "Route",
  "FeatureTailNum AS TailNum",
  "COALESCE(EngineManufacturer, 'Empty') AS EngineManufacturer",
  "COALESCE(EngineModel, 'Empty') AS EngineModel",
  "COALESCE(Manufacturer, 'Empty') AS Manufacturer",
```

仅供非商业用途或交流学习使用

```
    "COALESCE(ManufacturerYear, 'Empty') AS ManufacturerYear",
    "COALESCE(Model, 'Empty') AS Model",
    "COALESCE(OwnerState, 'Empty') AS OwnerState"
  )
```

最后，我们把最终输出保存到一个新的路径下，要记得：修改模型训练的脚本是指向改动后的新路径的：

```
# 存储为单个 JSON 文件
output_path = "{}/data/simple_flight_delay_features_airplanes.json".format(
  base_path
)
sorted_features.repartition(1).write.mode("overwrite").json(output_path)

# 复制部分文件到 JSON 行文件中
combine_cmd = \
  "cp {}/part* {}/data/simple_flight_delay_features_airplanes.jsonl".format(
  output_path,
  base_path
)
os.system(combine_cmd)
```

现在，我们可以把这些特征纳入模型中了。

在分类器模型中纳入飞机特征

现在，我们需要创建把新的飞机特征纳入分类器模型的脚本。代码如 *ch09/spark_model_with_airplanes.py* 所示，这是我们从 *ch09/improved_spark_mllib_model.py* 中复制并修改出来的。

首先，我们需要在读取训练数据时把增加的这几列也读取出来，包括 Route（通过 *ch09/extract_features_with_airplanes.py* 求得）：

```
schema = StructType([
  StructField("ArrDelay", DoubleType(), True),
  StructField("CRSArrTime", TimestampType(), True),
  StructField("CRSDepTime", TimestampType(), True),
  StructField("Carrier", StringType(), True),
  StructField("DayOfMonth", IntegerType(), True),
  StructField("DayOfWeek", IntegerType(), True),
  StructField("DayOfYear", IntegerType(), True),
  StructField("DepDelay", DoubleType(), True),
  StructField("Dest", StringType(), True),
  StructField("Distance", DoubleType(), True),
  StructField("FlightDate", DateType(), True),
  StructField("FlightNum", StringType(), True),
  StructField("Origin", StringType(), True),
  StructField("Route", StringType(), True),
  StructField("TailNum", StringType(), True),
```

仅供非商业用途或交流学习使用

```
  StructField("EngineManufacturer", StringType(), True),
  StructField("EngineModel", StringType(), True),
  StructField("Manufacturer", StringType(), True),
  StructField("ManufacturerYear", StringType(), True),
  StructField("OwnerState", StringType(), True),
])

input_path = "{}/data/simple_flight_delay_features_airplanes.json".format(
  base_path
)
features = spark.read.json(input_path, schema=schema)
features.first()
```

因为我们是通过左外连接加入新特征的，所以我们需要知道：结果中有多少条训练记录在连接时产生了空值。空值会导致对该字段进行索引的 StringIndexer 发生崩溃，因此我们修改了特征提取代码，显式移除了空值。到这里，数据中已经不会有空值了，所以如果还有空值的话，我们要输出一个记录空值的表格：

```
#
# 在使用 Spark ML 之前检查特征中是否存在空值
#
null_counts = [( \
  column, features_with_hour.where(
    features_with_hour[column].isNull()).count()) \
      for column in features_with_hour.columns]
cols_with_nulls = filter(lambda x: x[1] >0, null_counts)
print("\nNull Value Report")
print("-----------------")
print(tabulate(cols_with_nulls, headers=["Column", "Nulls"]))
```

接下来，我们和平常一样添加当日时间的小时字段，并把到达延误字段 ArrDelay 分桶，获取对应的 ArrDelayBucket。然后，我们需要把所有的字符串列转为索引值，包括新的飞机特征字段：

```
#
# pyspark.ml.feature 中的特征提取工具
#
from pyspark.ml.feature import StringIndexer, VectorAssembler

# 把分类字段转为索引值
string_columns = ["Carrier", "Origin", "Dest", "Route",
                  "TailNum", "EngineManufacturer",
                  "EngineModel", "Manufacturer",
                  "ManufacturerYear", "Owner",
                  "OwnerState"]
for column in string_columns:
  string_indexer = StringIndexer(
    inputCol=column,
    outputCol=column + "_index"
  )
```

仅供非商业用途或交流学习使用

```
string_indexer_model = string_indexer.fit(ml_bucketized_features)
ml_bucketized_features = string_indexer_model.transform(
  ml_bucketized_features
)

# 保存管道模型
string_indexer_output_path = \
  "{}/models/string_indexer_model_3.0.{}.bin".format(
  base_path,
  column
  )
  string_indexer_model.write().overwrite().save(string_indexer_output_path)
```

接下来，我们需要创建新的 VectorAssembler 来把所有特征组合到一个特征向量中，作为 Features_vec 列。和之前一样，索引值字段的字段名都是以原始字段名加上 _index 后缀来命名的。这一次，我们使用列表推导式来得到索引值字段的字段名：

```
# 把连续数值字段和称名字段索引组合到一个特征向量里
numeric_columns = [
  "DepDelay", "Distance",
  "DayOfMonth", "DayOfWeek",
  "DayOfYear", "CRSDepHourOfDay",
  "CRSArrHourOfDay"]
index_columns = [column + "_index" for column in string_columns]

vector_assembler = VectorAssembler(
  inputCols=numeric_columns + index_columns,
  outputCol="Features_vec"
)
final_vectorized_features = vector_assembler.transform(ml_bucketized_features)

# 保存数值向量组合器
vector_assembler_path = "{}/models/numeric_vector_assembler_4.0.bin".format(
base_path
)
vector_assembler.write().overwrite().save(vector_assembler_path)
```

其余的代码和 *ch09/improved_spark_mllib_model.py* 中的一样。要在新一轮迭代实验中测试新特征，只要运行：

```
python ch09/spark_model_with_airplanes.py .
```

注意：在第一次运行时模型崩溃了，因为我们需要把 maxBins 参数增加到 4896 才能用于新加的字段。改好之后，脚本就能正常运行了。让我们看看结果如何：

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

```
Experiment Log
--------------
Metric                 Average          STD
-----------------  ---------  -----------
accuracy            0.594262  0.000256266
weightedPrecision   0.513819  9.43221e-05
weightedRecall      0.594262  0.000256266
f1                  0.522066  0.000348499

Experiment Report
-----------------
Metric                     Score
-----------------  ------------
accuracy           -0.000394734
weightedPrecision  -0.127719
weightedRecall     -0.000394734
f1                 -0.00123417

Feature Importances
-------------------
Name                        Importance
------------------------  ------------
DepDelay                   0.859874
Route_index                0.0628756
CRSArrHourOfDay            0.0166273
Distance                   0.013517
TailNum_index              0.012384
Origin_index               0.011521
Dest_index                 0.00709832
Carrier_index              0.00695651
DayOfYear                  0.00416167
CRSDepHourOfDay            0.00218328
OwnerState_index           0.00144406
Manufacturer_index         0.00061876
EngineManufacturer_index   0.000421311
EngineModel_index          0.000311177
ManufacturerYear_index     4.56816e-06
DayOfWeek                  1.26539e-06
DayOfMonth                 0

Feature Importance Delta Report
-------------------------------
Feature                           Delta
------------------------  ------------
TailNum_index              0.012384
CRSArrHourOfDay            0.00680138
Route_index                0.00298732
Carrier_index              0.00246828
OwnerState_index           0.00144406
CRSDepHourOfDay            0.000911583
Manufacturer_index         0.00061876
Distance                   0.000527296
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

```
EngineManufacturer_index   0.000421311
EngineModel_index          0.000311177
DayOfYear                  4.07316e-05
ManufacturerYear_index     4.56816e-06
DayOfMonth                -2.07392e-07
DayOfWeek                 -6.7965e-05
Origin_index              -0.000563111
Dest_index                -0.00167737
DepDelay                  -0.0266118
```

看起来我们是白费力气了——实际上，模型的预测质量还下降了！唯一的优点是，增加的 TailNum 在特征重要性中的分数为 0.012。显然一些飞机比别的一些飞机更容易晚点，但没有很好地从我们尝试的飞机属性上体现出来。

让我们尝试仅仅选择纳入 TailNum 字段，看看分数有怎样的变化。同时我们可以顺便去掉 DayOfMonth 和 DayOfWeek 字段，因为它们看起来压根没起什么作用。我们只需把这些字段从 StringIndexer 的映射关系中剔除即可：

```
# 把分类字段转为索引值
string_columns = ["Carrier", "Origin", "Dest", "Route",
                  "TailNum"]
for column in string_columns:
  string_indexer = StringIndexer(
    inputCol=column,
    outputCol=column + "_index"
  )

  string_indexer_model = string_indexer.fit(ml_bucketized_features)
  ml_bucketized_features = string_indexer_model.transform(
    ml_bucketized_features)

  # 保存管道模型
  string_indexer_output_path = \
    "{}/models/string_indexer_model_4.0.{}.bin".format(
    base_path,
    column
  )
  string_indexer_model.write().overwrite().save(string_indexer_output_path)

# 把连续数值字段和称名字段索引组合到一个特征向量里
numeric_columns = [
  "DepDelay", "Distance",
  "DayOfYear",
  "CRSDepHourOfDay",
  "CRSArrHourOfDay"]
index_columns = [column + "_index" for column in string_columns]

  vector_assembler = VectorAssembler(
    inputCols=numeric_columns + index_columns,
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

```
    outputCol="Features_vec"
)
final_vectorized_features = vector_assembler.transform(ml_bucketized_features)

# 保存数值向量组合器
vector_assembler_path = \
  "{}/models/numeric_vector_assembler_5.0.bin".format(base_path)
vector_assembler.write().overwrite().save(vector_assembler_path)
```

这对分数起了正面作用，不过效果有限：准确率仅提高了 0.00031884。尽管如此，这还是说明现在我们所有的特征对模型的预测质量都有很大作用，而这正是我们想要看到的：

```
Feature Importances
-------------------
Name                Importance
---------------  ------------
DepDelay            0.879979
Route_index         0.0575757
Distance            0.0174215
TailNum_index       0.0120175
CRSArrHourOfDay     0.0117084
Origin_index        0.00789092
Carrier_index       0.00457943
DayOfYear           0.00408886
Dest_index          0.00311852
CRSDepHourOfDay     0.00161978
```

请牢记：对预测模型来说，越简单越好。如果一个特征不能对预测准确度产生足够大的影响，那就把它删掉。模型质量会提高，在生产环境中模型跑得更快，你也更容易理解添加的特征对模型的影响。简单的模型也更不容易受偏见的影响。

纳入飞行时间

还有一个我们没有考虑的事情就是飞行时间。用降落时间减去起飞时间，就可以求出飞行时间。由于飞行距离是前三位的特征，当天时间也会影响预测结果，因此飞行时间似乎能让预测质量更上一层楼。让我们试试看！

为了求出到达时间和起飞时间之间的差值，我们需要把这两个字段转为 UNIX 时间戳类型，用自 1970 年 1 月 1 日以来的秒数来表示时间。好在 Spark SQL 函数 `unix_timestamp` 刚好可以满足我们的需求。

参见 *ch09/extract_features_with_flight_time.py*，这是我们从 *ch09/extract_features_with_airplanes.py* 中复制并修改过的。我们只需要修改一行代码，也就是 `selectExpr`，加上飞行时间 `FlightTime` 字段的计算：

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

```
features_with_airplanes = features_with_airplanes.selectExpr(
  "FlightNum",
  "FlightDate",
  "DayOfWeek",
  "DayOfMonth",
  "DayOfYear",
  "Carrier",
  "Origin",
  "Dest",
  "Distance",
  "DepDelay",
  "ArrDelay",
  "CRSDepTime",
  "CRSArrTime",
  "Route",
  "FeatureTailNum AS TailNum",
  "COALESCE(EngineManufacturer, 'Empty') AS EngineManufacturer",
  "COALESCE(EngineModel, 'Empty') AS EngineModel",
  "COALESCE(Manufacturer, 'Empty') AS Manufacturer",
  "COALESCE(ManufacturerYear, 'Empty') AS ManufacturerYear",
  "COALESCE(Model, 'Empty') AS Model",
  "COALESCE(OwnerState, 'Empty') AS OwnerState",
  "unix_timestamp(CRSArrTime) - unix_timestamp(CRSDepTime) AS FlightTime"
)
```

我始终很喜欢 SQL 的强大功能，而 Spark 在这方面做得很好。Spark 还提供了数据流编程，可以和 SQL 结合起来使用，这样 Spark 成了可用的最佳编程模型。让我们把 `FlightTime` 字段纳入模型中。

参见 *ch09/spark_model_with_flight_time.py*，这是我们从 *ch09/spark_model_with_airplanes.py* 中复制并修改过的。我们要把 `FlightTime` 字段加到我们的 `StructType` 中并修改输入路径。然后我们还要在 `VectorAssembler` 中添加这个字段，因为这是个数值类型的字段，可以被直接组合到特征向量中：

```
schema = StructType([
  StructField("ArrDelay", DoubleType(), True),
  StructField("CRSArrTime", TimestampType(), True),
  StructField("CRSDepTime", TimestampType(), True),
  StructField("Carrier", StringType(), True),
  StructField("DayOfMonth", IntegerType(), True),
  StructField("DayOfWeek", IntegerType(), True),
  StructField("DayOfYear", IntegerType(), True),
  StructField("DepDelay", DoubleType(), True),
  StructField("Dest", StringType(), True),
  StructField("Distance", DoubleType(), True),
  StructField("FlightDate", DateType(), True),
  StructField("FlightNum", StringType(), True),
  StructField("Origin", StringType(), True),
  StructField("Route", StringType(), True),
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

```
    StructField("TailNum", StringType(), True),
    StructField("EngineManufacturer", StringType(), True),
    StructField("EngineModel", StringType(), True),
    StructField("Manufacturer", StringType(), True),
    StructField("ManufacturerYear", StringType(), True),
    StructField("OwnerState", StringType(), True),
    StructField("FlightTime", IntegerType(), True)
  ])

  input_path = "{}/data/simple_flight_delay_features_flight_times.json".format(
    base_path
  )
  features = spark.read.json(input_path, schema=schema)
  features.first()

...

  # 把连续数值字段和称名字段索引值组合到单个特征向量中
  numeric_columns = [
    "DepDelay", "Distance",
    "DayOfYear",
    "CRSDepHourOfDay",
    "CRSArrHourOfDay",
    "FlightTime"]
  index_columns = [column + "_index" for column in string_columns]

  vector_assembler = VectorAssembler(
    inputCols=numeric_columns + index_columns,
    outputCol="Features_vec"
  )
  final_vectorized_features = vector_assembler.transform(ml_bucketized_features)
```

现在我们可以测试新模型了：

```
python ch09/spark_model_with_flight_time.py .
```

输出显示预测模型的表现获得了明显提高！weightedPrecision 提高了 0.12，FlightTime 占特征重要性约 0.5%。还要注意引入 FlightTime 特征也导致了 Distance 和 DepDelay 特征重要性的下降，这应该是符合预期的：距离特征 Distance 和飞行时间 FlightTime 在概念上是相似的，而 DepDelay 本来就是最重要的特征。总之，模型表现和特征重要性指标都表明引入 FlightTime 特征对于改进模型是有价值的：

```
Experiment Report
-----------------
Metric                  Score
-----------------  ----------
accuracy           0.00124616
weightedPrecision  0.117773
weightedRecall     0.00124616
f1                 0.00453277
```


仅供非商业用途或交流学习使用

```
Feature Importances
-------------------
Name                Importance
---------------  ------------
DepDelay            0.860049
Route_index         0.0742784
CRSArrHourOfDay     0.0135059
Origin_index        0.0123399
TailNum_index       0.0120064
Distance            0.00649571
Carrier_index       0.00563587
DayOfYear           0.00479174
FlightTime          0.00452475
CRSDepHourOfDay     0.00378075
Dest_index          0.00259198
Feature Importance Delta Report
-------------------------------
Feature                  Delta
---------------  ------------
Route_index        0.0167027
FlightTime         0.00452475
Origin_index       0.00444897
CRSDepHourOfDay    0.00216097
CRSArrHourOfDay    0.00179746
Carrier_index      0.00105644
DayOfYear          0.00070288
TailNum_index     -1.10573e-05
Dest_index        -0.00052654
Distance          -0.0109258
DepDelay          -0.0199308
```

到这里，我们又一次感觉到似乎已经挖完了时间 / 日期特征的各种可能性（至少是在不借助我能力范围外的复杂时间序列分析技术的前提下）。

本章小结

在本章中，我们介绍了如何使用收集的数据改进我们的模型。我们可以将这种方法结合到应用程序的部署中，不断改进我们的预测系统。

仅供非商业用途或交流学习使用

附录 A

安装手册

在本附录中，我们会具体介绍如何安装本书中用到的软件栈。

安装 Hadoop

Hadoop 的最新版本可以从 Hadoop 官方网站的下载页面（*http://hadoop.apache.org/releases.html*）下载到。在本书写作时，Hadoop 的最新版本为 2.7.3，而当你读到此处的时候，很有可能已经有了更新的版本。

本书提供了一份包含 Hadoop 安装的一键安装脚本 *manual_install.sh*。除了下载和解压 Hadoop，我们还要配置好 Hadoop 相关的环境变量（HADOOP_HOME、HADOOP_CLASSPATH 以及 HADOOP_CONF_DIR），并把 Hadoop 的可执行文件路径配置到 PATH 中。首先，设好 PROJECT_HOME 变量，这有助于我们轻松地找到正确的路径。你需要通过编辑 *.bash_profile* 文件来自行设定这个变量：

```
export PROJECT_HOME=/Users/rjurney/Software/Agile_Data_Code_2
```

接下来我们就能自动完成整个环境的设置了。下面是一键安装脚本 *manual_install.sh* 中的相关部分：

```
# 需要修改这个链接指向最新的 Hadoop 版本，请参阅 http://hadoop.apache.org/releases.html
curl -Lko /tmp/hadoop-2.7.3.tar.gz \
http://apache.osuosl.org/hadoop/common/hadoop-2.7.3/hadoop-2.7.3.tar.gz

mkdir hadoop
tar -xvf /tmp/hadoop-2.7.3.tar.gz -C hadoop --strip-components=1
echo '# Hadoop environment setup' >> ~/.bash_profile
export HADOOP_HOME=$PROJECT_HOME/hadoop
echo 'export HADOOP_HOME=$PROJECT_HOME/hadoop' >> ~/.bash_profile
```


仅供非商业用途或交流学习使用

```
export PATH=$PATH:$HADOOP_HOME/bin
echo 'export PATH=$PATH:$HADOOP_HOME/bin' >> ~/.bash_profile
export HADOOP_CLASSPATH=$(hadoop classpath)
echo 'export HADOOP_CLASSPATH=$(hadoop classpath)' >> ~/.bash_profile
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
echo 'export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop' >> ~/.bash_profile
```

安装 Spark

在写作本书时，Spark 的最新版本是 2.1.0。你可以参考 Spark 官方文档（*http://spark.apache.org/docs/latest/*）在自己的电脑上安装 Spark，也可以使用我们提供的一键安装脚本 *manual_install.sh*：

```
# 需要修改此链接，请参阅 http://spark.apache.org/downloads.html
curl -Lko /tmp/spark-2.1.0-bin-without-hadoop.tgz \
  http://d3kbcqa49mib13.cloudfront.net/spark-2.1.0-bin-without-hadoop.tgz

mkdir spark
tar -xvf /tmp/spark-2.1.0-bin-without-hadoop.tgz -C spark --strip-components=1
echo ""> ~/.bash_profile
echo "# Spark environment setup"> ~/.bash_profile
export SPARK_HOME=$PROJECT_HOME/spark
echo 'export SPARK_HOME=$PROJECT_HOME/spark' >> ~/.bash_profile
export HADOOP_CONF_DIR=$PROJECT_HOME/hadoop/etc/hadoop/
echo 'export HADOOP_CONF_DIR=$PROJECT_HOME/hadoop/etc/hadoop/' >> ~/.bash_profile
export SPARK_DIST_CLASSPATH=`$HADOOP_HOME/bin/hadoop classpath`
echo 'export SPARK_DIST_CLASSPATH=`$HADOOP_HOME/bin/hadoop classpath`' >> \
   ~/.bash_profile
export PATH=$PATH:$SPARK_HOME/bin
echo 'export PATH=$PATH:$SPARK_HOME/bin' >> ~/.bash_profile

# 在 Spark 本地模式中需要配置 spark.io.compression.codec
cp spark/conf/spark-defaults.conf.template spark/conf/spark-defaults.conf
echo 'spark.io.compression.codec org.apache.spark.io.SnappyCompressionCodec' >>
  spark/conf/spark-defaults.conf

# 为 Spark 分配 8 GB 的内存
echo "spark.driver.memory 8g"> $SPARK_HOME/conf/spark-defaults.conf

echo "PYSPARK_PYTHON=python3"> $SPARK_HOME/conf/spark-env.sh
echo "PYSPARK_DRIVER_PYTHON=python3"> $SPARK_HOME/conf/spark-env.sh

# 配置 log4j 参数减少日志输出
cp $SPARK_HOME/conf/log4j.properties.template $SPARK_HOME/conf/log4j.properties
sed -i .bak 's/INFO/ERROR/g' $SPARK_HOME/conf/log4j.properties
```

请注意这里下载的 URL 是动态变化的，你可以从 Spark 下载页面（*http://spark.apache.org/downloads.html*）获取当前的 URL 用于在控制台中的安装操作。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

安装 MongoDB

安装 MongoDB 的说明可以从官网找到，官网上还提供了一份很好的教程。我建议在动手前先参考一下这些材料。

从 MongoDB 的下载中心页面（*http://www.mongodb.org/downloads*）上，可以下载到适用于各操作系统的最新版 MongoDB。然后用如下指令安装 MongoDB：

```
curl -Lko /tmp/$MONGO_FILENAME $MONGO_DOWNLOAD_URL
mkdir mongodb
tar -xvf /tmp/$MONGO_FILENAME -C mongodb --strip-components=1
export PATH=$PATH:$PROJECT_HOME/mongodb/bin
echo 'export PATH=$PATH:$PROJECT_HOME/mongodb/bin' >> ~/.bash_profile
mkdir -p mongodb/data/db
```

现在启动 MongoDB 服务器端：

```
mongodb/bin/mongod --dbpath mongodb/data/db &
```

电脑关机重启后，需要再次执行这条命令来启动 MongoDB。现在让我们先打开 Mongo 命令行工具，获取帮助信息：

```
mongob/bin/mongo --eval help
```

最后，让我们通过插入一条记录来创建一个集合，然后把这条记录查询出来：

```
>db.test_collection.insert(
{'name': 'Russell Jurney', 'email': 'russell.jurney@gmail.com'})

WriteResult({ "nInserted": 1 })

>db.test_collection.findOne({'name': 'Russell Jurney'})
{
        "_id": ObjectId("56f20fa811a5b44cf943313c"),
        "name": "Russell Jurney",
        "email": “russell.jurney@gmail.com"
}
>
```

现在，我们已经安装好 MongoDB，并且简单地使用过它了。

安装 MongoDB 的 Java 驱动

你还需要安装 MongoDB 的 Java 驱动。在写作本书时，最新的稳定版本为 3.4.2。你可以使用 `curl` 命令来安装驱动程序，如下所示：

```
curl -Lko lib/mongo-java-driver-3.4.2.jar \
   http://central.maven.org/maven2/org/mongodb/mongo-java-driver/3.4.0/ \
      mongo-java-driver-3.4.0.jar
```

仅供非商业用途或交流学习使用

安装 mongo-hadoop

mongo-hadoop（*https://github.com/mongodb/mongo-hadoop*）项目是用来连接 Hadoop 和 Spark 与 MongoDB 的。你可以从它的发布页面（*https://github.com/mongodb/mongo-hadoop/releases*）进行下载。

编译 mongo-hadoop

你需要使用自带的 gradlew 命令编译这个项目，然后把编译得到的 JAR 包复制到 *lib/* 目录里去：

```
# 在项目根目录的 mongo-hadoop 目录中安装 mongo-hadoop 项目
curl -Lko /tmp/r1.5.2.tar.gz \
  https://github.com/mongodb/mongo-hadoop/archive/r1.5.2.tar.gz
mkdir mongo-hadoop
tar -xvzf /tmp/r1.5.2.tar.gz -C mongo-hadoop --strip-components=1
# 下面编译 mongo-hadoop-spark 的 jar 包
cd mongo-hadoop
./gradlew jar
cd ..
cp mongo-hadoop/spark/build/libs/mongo-hadoop-spark-*.jar lib/
cp mongo-hadoop/build/libs/mongo-hadoop-*.jar lib/
```

安装 pymongo_spark

接下来，我们需要安装 pymongo_spark 包，它可以让我们从 PySpark 中用一条简单的命令将数据存储到 Mongo 中。pymongo_spark 包含在 mongo-hadoop 项目中：

```
# 编译 pymongo_spark 包
cd mongo-hadoop/spark/src/main/python
python setup.py install
cd $PROJECT_HOME
cp mongo-hadoop/spark/src/main/python/pymongo_spark.py lib/
export PYTHONPATH=$PYTHONPATH:$PROJECT_HOME/lib
echo 'export PYTHONPATH=$PYTHONPATH:$PROJECT_HOME/lib' >> ~/.bash_profile
```

安装 Elasticsearch

Elasticsearch 在官网上提供了一份很不错的教程。从官方下载页面下载（*http://www.elastic.co/downloads*），然后使用下面的命令安装 Elasticsearch：

```
curl -Lko /tmp/elasticsearch-2.3.5.tar.gz \
   https://download.elastic.co/elasticsearch/release/org/elasticsearch/ \
     distribution/tar/elasticsearch/2.3.5/elasticsearch-2.3.5.tar.gz
mkdir elasticsearch
```

仅供非商业用途或交流学习使用

```
tar -xvzf /tmp/elasticsearch-2.3.5.tar.gz -C elasticsearch -strip-components=1
```

用下面的命令来运行 Elasticsearch:

```
elasticsearch/bin/elasticsearch 2>1 > /dev/null &
```

这就对了，本地模式的引擎已经成功启动！注意，在关机或重启后，你需要重新运行这条命令才能启动 Elasticsearch。使用 `curl` 可以很容易地在 Elasticsearch 中插入一条记录并查询：

```
curl -XPUT 'localhost:9200/customer/external/1?pretty' -d '
{
  "name": "Russell Jurney"
}'

curl 'localhost:9200/customer/_search?q=*&pretty'
```

下面是这条查询语句的返回结果：

```
{
  "took": 81,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 1,
    "max_score": 1.0,
    "hits":[{
      "_index": "customer",
      "_type": "external",
      "_id": "1",
      "_score": 1.0,
      "_source": {
        "name": "Russell Jurney"
      }
      }]
    }
  }
```

安装 Elasticsearch 的 Hadoop 支持库

你可以从 ES-Hadoop 的下载页面（*https://www.elastic.co/downloads/hadoop*）下载到 Elasticsearch 的 Hadoop 支持库，然后使用下面的命令安装：

```
# 安装 Elasticsearch 的 Hadoop 支持库
curl -Lko /tmp/elasticsearch-hadoop-5.0.0-alpha5.zip \
  http://download.elastic.co/hadoop/elasticsearch-hadoop-5.0.0-alpha5.zip
```

仅供非商业用途或交流学习使用

```
unzip /tmp/elasticsearch-hadoop-5.0.0-alpha5.zip
mv elasticsearch-hadoop-5.0.0-alpha5 elasticsearch-hadoop
cp elasticsearch-hadoop/dist/elasticsearch-hadoop-5.0.0-alpha5.jar lib/
cp elasticsearch-hadoop/dist/elasticsearch-spark-20_2.10-5.0.0-alpha5.jar lib/
echo "spark.speculation false">> $PROJECT_HOME/spark/conf/spark-defaults.conf
```

配置我们的 Spark 环境

每次我们从命令行启动 pyspark 都需要设置 Mongo 和 Elasticsearch 相关的 JAR 包参数，这样太麻烦。所幸我们可以使用 Spark 的配置文件——*spark/conf/spark-defaults.conf* 来解决这个问题，实现自动加载 JAR 包。具体来讲，环境变量 `spark.jars` 可以处理读取 JAR 包。

在 *manual_install.sh* 中，我们是这样做的：

```
# 为 Spark 配置 Mongo 和 Elasticsearch 的 JAR 包
echo "spark.jars $PROJECT_HOME/lib/mongo-hadoop-spark-2.0.0-rc0.jar,\
$PROJECT_HOME/lib/mongo-java-driver-3.2.2.jar,\
$PROJECT_HOME/lib/mongo-hadoop-2.0.0-rc0.jar,\
$PROJECT_HOME/lib/elasticsearch-spark-20_2.10-5.0.0-alpha5.jar,\
$PROJECT_HOME/lib/snappy-java-1.1.2.6.jar,\
$PROJECT_HOME/lib/lzo-hadoop-1.0.0.jar" \
>> spark/conf/spark-defaults.conf
```

这样做好之后，我们就只需要运行 `PYSPARK_DRIVER_PYTHON=ipython pyspark` 来启动 PySpark 了。

安装 Kafka

在写作本书时，Kafka 最新的稳定版本是 0.10.2.0。你可以在 Kafka 官方下载页面（*https://kafka.apache.org/downloads*）上获取最新的稳定版本，然后按如下命令进行安装（如果有必要的话，请修改替换掉脚本中的版本号）：

```
# 安装 Apache Kafka
curl -Lko /tmp/kafka_2.11-0.10.2.0.tgz \
  http://www-us.apache.org/dist/kafka/0.10.2.0/kafka_2.11-0.10.2.0.tgz
mkdir kafka
tar -xvzf /tmp/kafka_2.11-0.10.2.0.tgz -C kafka --strip-components=1
```

这样，Kafka 就装好了。注意这里用的是本地模式，而 Kafka 的初衷当然是可以运行在几乎各种规模的分布式环境里。使用本地模式进行开发非常方便，而开发完成后只需要直接部署在分布式模式下就可以了。

安装 scikit-learn

在 Anaconda（*https://docs.continuum.io/anaconda/*）中预装了 `scikit-learn`，因此，如果你

仅供非商业用途或交流学习使用

安装了 Anaconda，scikit-learn 就已经安装好了。如果没有 Anaconda，那么你需要自己安装 sklearn。

你可以使用 pip 进行安装 sklearn:

```
pip install sklearn
```

当然也可以用 easy_install 安装：

```
easy_install sklearn
```

安装 sklearn 可能需要你先装好 numpy 和 scipy（*https://www.scipy.org/scipylib/download.html*）这两个科学计算的库。

你可以通过官方提供的优秀教程熟悉 sklearn 。

安装 Zeppelin

在写作本书时，Apache Zeppelin 的最新版本为 0.6.2。你可以从 Zeppelin 官方下载页面（*https://zeppelin.apache.org/download.html*）找到最新版本，然后修改下面脚本中的版本号。安装 Zeppelin 非常简单。同样，请参考 *manual_install.sh* 中对应的部分：

```
# 安装 Apache Zeppelin
curl -Lko /tmp/zeppelin-0.6.2-bin-all.tgz \
  http://www-us.apache.org/dist/zeppelin/zeppelin-0.6.2/ \
    zeppelin-0.6.2-bin-all.tgz
mkdir zeppelin
tar -xvzf /tmp/zeppelin-0.6.2-bin-all.tgz -C zeppelin —strip-components=1

# 配置 Zeppelin
cp zeppelin/conf/zeppelin-env.sh.template zeppelin/conf/zeppelin-env.sh
echo "export SPARK_HOME=$PROJECT_HOME/spark">> zeppelin/conf/zeppelin-env.sh
echo "export SPARK_MASTER=local">> zeppelin/conf/zeppelin-env.sh
echo "export SPARK_CLASSPATH=">> zeppelin/conf/zeppelin-env.sh
```

运行 zeppelin/bin/zeppelin-daemon.sh start 来启动 Zeppelin，然后打开浏览器访问 *http://localhost:8080* 查看用户界面。推荐你在安装后按照 Zeppelin 的官方教程学做一遍。

仅供非商业用途或交流学习使用

关于作者

Russell Jurney 在赌场游戏中提高了数据分析的技能，构建了网络应用程序来分析美国和墨西哥的老虎机的表现。在涉足创业、互动媒体、记者等行业后，他搬到硅谷，在 Ning 和 LinkedIn 从事构建分析型应用的工作。Russell 现在是 Data Syndrome 的首席顾问，他帮助公司使用本书所介绍的原则和方法构建分析型产品。

仅供非商业用途或交流学习使用

封面介绍

《Spark 全栈数据分析》的封面动物是银绢猴（*Mico argentatus*）。这种小型的美洲猴子居住在亚马逊雨林东部地区以及巴西。尽管名字听起来银绢猴是银色的，但其实它的颜色从近白色到深褐色都有。褐色的银绢猴有无毛的耳朵和脸，有时被称为赤耳狨。银绢猴平均可以长到 22 厘米，大小和松鼠差不多，因此它们在树冠和茂盛的植物间行动非常容易。银绢猴的族群一般由 12 个左右的个体组成，所有成员都会帮助照顾幼崽。雄性银绢猴白天将幼崽带在身边，每两到三个小时回到雌性银绢猴那里喂食幼崽。幼崽在大约六个月时断奶，在大约一到两岁时成年。银绢猴的食物主要为树汁和树胶。它们使用锋利的牙齿在树上挖洞吸食树汁，偶尔也会吃水果、树叶以及昆虫。然而随着热带雨林不断被砍伐，银绢猴已经开始吃人类种植的谷物了，因此很多农民把它们视为害兽。农业领域正在对它们实施大规模灭绝方案，对于银绢猴整体种群数量的影响尚未探明。由于体积小而且性格温和，因此银绢猴经常被用作医学研究的对象。对银绢猴的受精、胎盘发育以及胚胎干细胞的研究可能揭示人类发育问题和遗传疾病的原因。除了实验室，银绢猴在动物园也很受欢迎，因为它们是昼行动物（白天活跃），而且充满活力；它们长长的爪子意味着它们可以在树上快速移动，而且雄性和雌性个体间都会高声交流。

O'Reilly 丛书封面上的许多动物都濒临灭绝，而它们对这个世界来说都很重要。要了解更多你力所能及的事，请访问 *animals.oreilly.com*。

封面图片来自 Lydekker 所著的 Royal Natural History。

O'REILLY®

Spark全栈数据分析

数据科学团队在寻求把科研成果转化为有意义的数据科学应用时，不仅仅需要使用合适的工具，还应该使用合适的方法，才能获得成功。有了这本修订过的基于Spark的实战指南，初出茅庐的数据科学家们将学到如何使用敏捷数据科学的开发方法论，使用Python、Apache Spark、Kafka及其他工具构建数据应用。

作者Russell Jurney展示了如何使用Apache Kafka、Apache Spark、MongoDB、Elasticsearch、D3.js、scikit-learn以及Apache Airflow组合成的数据平台，构建、部署、完善分析型应用程序。你会学到一种迭代的方法，让你能够根据数据所示快速改变分析类型，把数据科学工作以网络应用程序的形式发布，对所在机构产生有价值的影响。

- 在一系列敏捷冲刺中根据数据价值金字塔模型从数据中创造价值。
- 从多个数据集中提取特征构建统计模型。
- 通过图表进行数据可视化，通过交互式报表展示数据的各种维度。
- 通过分类和回归，使用历史数据预测未来。
- 把预测结果带入实际行动。
- 在每个冲刺结束后收集用户反馈，让项目始终向正确的方向发展。

Russell Jurney在博彩游戏中练就了数据分析的技能，构建了网络应用程序来分析美国和墨西哥的博彩机器的表现。在涉足创业、互动媒体、记者等行业后，他搬到硅谷，在Ning和LinkedIn从事构建分析型应用的工作。Russell现在是Data Syndrome的首席顾问，他帮助公司使用本书所介绍的原则和方法构建分析型产品。

图书分类：数据分析

策划编辑：刘恩惠
责任编辑：牛勇

Broadview®
WWW.BROADVIEW.COM.CN

O'Reilly Media, Inc.授权电子工业出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale in the mainland of China (excluding Hong Kong, Macao SAR and Taiwan)

ISBN 978-7-121-35166-2

定价：99.00元